# FINAL REPORT

**for**

**Contract DAAB07-97-C-H501**

**Contract Line Item 0001AA**

**SBIR TOPIC A96-075**

## *VISUAL SOFTWARE DEVELOPMENT FOR PARALLEL MACHINES*

**March 31, 1997**



**Submitted To:**

## SOFTWARE ENGINEERING CENTER
## U.S. ARMY CECOM



**Submitted By:**

**PREDICTION SYSTEMS, INC.**
309 Morris Avenue
Spring Lake, NJ 07762

☎ (908)449-6800 (908)449-0897
psi@predictsys.com www.predictsys.com

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY (*Leave Blank*) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | 5/20/97 | FINAL |

**4. TITLE AND SUBTITLE**
"Visual Software Development for Parallel Machines"

**5. FUNDING NUMBERS**
CONTRACT
DAABO7-97-H01

**6. AUTHOR(S)**
Mr. William Cave, Mr. Robert Wassmer, and
Dr. Henry Ledgard

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Prediction Systems, Inc.
309 Morris Avenue. Suite G
Spring Lake, NJ 07762

**8. PERFORMING ORGANIZATION REPORT NUMBER**
PSI-97002

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Commander US Army CECOM
AMSEL-ACCB-C-BP
Fort Monmouth, NJ 07703-5008

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**
None

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
Unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** (*Maximum 200 words*)

If the benefits of parallel processing do not sufficiently exceed the cost to both develop the software and manage the allocation of processor resources during runtime, then commercialization, based upon economics, won't occur. These economic goals will be achieved if: (1) people who understand the problems to be solved can describe them easily and directly to the computer without concern for parallelism, or even prior knowledge of computer programming; and (2) the run-time software is generated automatically to take full effective advantage of the inherent parallelism of the problem on a parallel machine.

This Phase I effort shows that optimal allocation of processes to processors can result from an architecture that produces independent modules. When an architect follows PSI's visual design rules, this occurs automatically. Visualization is the result of separating data from instructions, allowing a one-to-one mapping of graphical icons into actual code. An architect can determine critical independence properties of a design just by visual inspection of engineering drawings. By following PSI's design rules, this same visual inspection of the drawings can be used to quickly assess the inherent parallelism of a system.

| 14. SUBJECT TERMS | | 15. NUMBER OF PAGES |
|---|---|---|
| Computer-Aided Design (CAD) | Simulation | |
| Software | Parallel Processing | 16. PRICE CODE |

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | Unlimited |

# TABLE OF CONTENTS


| SECTION | TITLE | PAGE |
|---|---|---|
| 1. | BACKGROUND | 1 |
| 2. | INTRODUCTION | 4 |
| 3. | TECHNICAL OBJECTIVES | 6 |
| 4. | OVERVIEW OF THE PROPOSED TECHNOLOGY | 7 |
| 5. | PHASE I ANALYSIS - DEFINITIONS | 19 |
| 6. | PHASE I ANALYSIS - MEASURES OF MERIT | 25 |
| 7. | DESIGN OF EXPERMENTS TO SUPPORT ANALYSIS | 27 |
| 8. | REVIEW OF FINDINGS | 31 |
| 9. | TIME DIFFERENTIAL CONCURRENT PROCESSING | 34 |
| 10. | CONCURRENT PROCESSING PROPERTIES OF SYSTEMS | 37 |
| 11. | REVIEW OF OBJECTIVES & PROPOSED APPROACH | 40 |
| 12. | OVERVIEW OF TECHNICAL APPROACH | 42 |
| 13. | RUN-TIME ARCHITECTURE CONSIDERATIONS | 45 |
| 14. | TOP-LEVEL RUN-TIME SYSTEM ARCHITECTURE | 53 |
| 15. | OPTIMAL PROCESSOR ALLOCATION METHODS | 55 |
| 16. | INTERPROCESSOR RESOURCE PROTOCAOL DESIGN | 57 |
| 17. | USER-INTERFACE DESIGN CONSIDERATIONS | 59 |
| 18. | USING THREADS TO SUPPORT PARALLEL PROCESSING | 62 |
| 19. | SUMMARY AND CONCLUSIONS | 66 |
| 20. | REFERENCES | 68 |

# 1. BACKGROUND

## HISTORIC APPROACHES

Hardware designers have succeeded in producing parallel and distributed processor computers with theoretical speeds well into the gigaflop range, see for example Frank, [1], Sietz, [2], Bell, [3], [4], and Daly, [5]. However, the practical use of these machines on all but some very special problems is extremely limited. The inability to use this power is due to great difficulties encountered when trying to translate real world problems into software that makes effective use of highly parallel machines.

Much of the parallel computing software work done to date has focused on increasing the speed of mathematical calculations, e.g., matrix inversion. Even in many of these applications, the best solution approaches do not easily lend themselves to parallel processing. For example, the fastest known algorithms for sparse matrix inversion are inherently sequential. This is because these methods, known as symbolic preprocessors, eliminate looping and testing, leaving only the minimum sequential set of add, subtract, multiply, and divide operations to be performed, see for example, Berry, [6], and Hachtel, [7]. In addition, focusing on parallelism in short instruction strings inside program loops does not necessarily lead to efficient use of large numbers of processors, since the overhead required to control which processor will perform what set of instructions using what data may take as long as the user instruction strings themselves, see Reiher [8]. This clearly depends upon the problem type as well as the solution approach.

To achieve efficient use of parallel computing resources, one must take effective advantage of the inherent parallelism of the problem to be solved. If the problem has little inherent parallelism, i.e., each step must follow in sequence, with each depending upon the prior outcome, then parallel processors will not help speed up the solution. If, on the other hand, large blocks of code can be processed at the same time independently, parallel processor computers should significantly improve the speed.

## COMMERCIAL MARKET REQUIREMENTS

The bottom line is that both the cost to develop the software for a parallel computer, and the cost of managing the allocation of parallel processor resources during runtime must not outweigh the benefits of their use. If the benefits do not sufficiently exceed these costs, then real commercialization, based upon solid economics, will not occur. These economic goals will be achieved if the following requirements can be met:

1. The people who understand the problems to be solved must be able to describe them easily and directly to the computer without concern for parallelism, or even prior knowledge of computer programming.

2. The software must be generated automatically so as to take full effective advantage of the inherent parallelism of the problem on a highly parallel computer.

These two requirements are tightly interrelated. The user should not be concerned whether the problem is being solved on a single processor machine, or one with hundreds of processors. The run-time software must be generated to make efficient use of the available parallelism of the host machine, adapting to changes in the environment, a very tedious but mechanical process.

## PREDICTION SYSTEMS, INC. - EXPERIENCE AND CAPABILITIES

Over the past thirty years, the principals of PSI have been immersed in developing CAD systems and large scale simulations to support engineering design and test. This work initially transformed graphic problem specification into continuous system simulation for electronic circuit design. This work was followed by discrete time simulation for modeling signal processing systems.

In the early 1980s, PSI was engaged by the Army to provide technical analysis of the design of a number of advanced communication systems. This was followed by engineering support of test planning and simulations for test augmentation. All of these efforts required very detailed models and simulations, typically built by the prime contractors. Two major problems were encountered by the Army. First, the simulations were extremely complex software efforts that, typically, were never completed properly. Second, the simulations that were completed took enormous computer resources and days to run, making their use virtually impractical.

Because of its experience in building simulation tools, PSI made a major business decision to invest in the development of CAD tools that would: (1) make it easy to build the large simulations required; and (2) run on a parallel processing machine to reduce running time.

Since 1982, PSI has focused on graphical CAD tools to support real-time control and communication system design. These tools are the General Simulation System (GSS) for large scale discrete event simulation, and the Virtual System Environment (VSE) for building software. These are open systems that are commercially available and used internationally.

Many millions of dollars worth of U.S. Army, Navy, and Air Force models and simulations currently reside in GSS, with the Government owning the source code. Some of these simulations are currently being used in very large Distributed Interactive Simulation (DIS) experiments. GSS and VSE are both built in VSE. The applications built with these systems represent a large class of problems that contain complex combinations of decision processes, as well as mathematics - all with significant inherent parallelism.

Using these CAD tools, PSI has accumulated a comprehensive database of large scale simulations and software for this class of problems. Most have excessive running times (some run for days). These systems were developed using the GSS and VSE graphic CAD technology that makes it very easy for engineers to build and support complex software without programming experience. The computer code (currently C) is generated automatically. This technology was described in Section E of PSI's Phase I proposal. Having analyzed potential solutions to the

excessive running time problem for the last eight years, it is our hypothesis that this class of problems can take full advantage of machines whose processors range from 100s to 1000s. More importantly, we have developed a unique and innovative extension to our technology that can make software development easier on parallel processors than on conventional computers while automatically generating the code. This approach clearly satisfies the two major requirements described in the above bullets.

# 2. INTRODUCTION

## UNDERSTANDING THE PROBLEM

Underlying our research on this project was the concern that no one has translated the power of parallel processing into solid economic benefits. As a result, commercial utilization has been virtually nonexistant to date. We wanted to ensure that we did not get caught in the magical attraction of potential speed multipliers that equate to the number of processors, a theory that has not been realistically applied in practice. We feel this has been a major flaw in past work in parallel processing. This concern bears no reflection on two, four, or even eight processor machines currently used commercially. These smaller machines are generally used to run independent tasks concurrently, tasks that have virtually no connections between them, and therefore have independent software architectures.

During the course of the Phase I research effort, PSI instrumented two of its largest simulations. These are the Multi-Switch Simulation (MSS) and the MIL-STD-188-220 simulation. Both of these simulations were built for DISA/JIEO at Ft. Monmouth, NJ. Various scenarios were run, but it did not take much data to turn some surprising results. To place these results in context so that they may be appreciated by readers unfamiliar with prior work by PSI, we will describe another simulation, the EPLRS Simulation Facility (ESF) built for PM TRCS at CECOM, Ft. Monmouth. Experiments that relate to parallel processing were performed using this facility during the ESF Verification and Validation efforts for PM TRCS in 1994-5.

As a result of our findings under this Phase I effort, we have addressed the problem of why no one has discovered the solution to effective use of parallel processors for a single task, e.g., a large simulation. In looking at this concern, we have uncovered a need to characterize systems in terms of properties that would lend themselves to effective use of parallel processing. These properties are described using comparisons with the EPLRS Simulation Facility. Having defined these properties, one can look at an application, including a simulation, and determine the applicability of parallel processing.

## OVERVIEW OF PSI's PROPOSED APPROACH

Given this new perspective on our own applications, and the new data we have gathered, an approach is proposed for making effective use of parallel processors. In simulation, it requires the design of model architectures that maximize independence. It also requires determinination of a $\Delta$Tmax for which concurrently scheduled processes can advance beyond the clock time of the earliest running process. This new approach applies to large simulations running on networks of computers, each with their own operating system (OS), e.g., those used in Distributed Interactive Simulation (DIS) as well as to the tightly coupled Shared Memory Programming (SMP) systems that can theoretically provide very high speed multipliers.

This proposed solution is not new at PSI. In the past, it has only been applied to simulations running on small networks of computers. We have now concluded that this approach is much more general, encompassing our previously proposed SMP approach (described in the Phase I proposal) as $\Delta$Tmax goes to zero.

To implement this modified approach, there are two elements required to achieve our overall project objectives. These are:

- A visual (graphical) user interface to designing module architectures that insures taking maximum advantage of module independence, and thus parallelism.
- An run-time environment that dynamically assigns parallel processors to processes as loading changes, while supporting a wide variety of problem types and machine environments.

PSI has developed a visual approach to designing software architecture that insures mapping the inherent parallelism of a system (or simulation) into independent modules (or models). What is now required are enhancements to the translation systems and run-time environments that will automate the mapping of independent modules or models into parallel processes.

During the course of the Phase I effort, we have addressed overall architectural issues associated with implementation of the required enhancements. To this end, we have specified an initial architecture for multiprocessor scheduling. We have also specified the protocols for communication of resource copies to insure memory coherency and synchronization of information among multiple distributed processor network architectures. We have used simulation as the example for covering design issues because this presents the more difficult and encompassing case for both real-time systems and simulations.

# 3. TECHNICAL OBJECTIVES

## PHASE I OBJECTIVES

The technical objectives described here are taken from PSI's proposal in response to SBIR Topic A96-075, Visual Software Development for Parallel Machines. This proposal was based upon PSI's graphical Computer-Aided Design (CAD) technology that provides users with an architecture environment to design independent modules and an architectural database that can support a parallel processor run-time environment. The objective of the Phase I research was to demonstrate how the architectural database could be used in a modified GSS run-time environment that optimizes the allocation of parallel processors to processes. In addition, we investigated certain modifications to the architecture environment to automate generation of independent module *instances* to run on separate processors.

Specifically, we have worked to determine what it will take to obtain a first-order average speed-up multiplier of 0.9N where N is the smaller of (1) the number of sufficiently complex independent modules, or (2) the number of parallel processors available. This implies getting a speed improvement of 90% when running on a parallel processor computer containing 100 processors if the number of independent module instances is greater than 100. As a result of our graphical module design rules, all large scale systems built by PSI contain hundreds of complex independent module instances.

Note that this is a first-order approach using the basic features of the proposed model development environment. Additional speed-up factors still remain below the top level modules, e.g., when submodules are instanced, or large processes are themselves independent.

## OVERVIEW OF THE PHASE II PLAN

The research performed in Phase I derived pertinent statistics from our large scale software database. These statistics were analyzed for pragmatic design decisions in preparation for Phase II. As part of our Phase I analysis, we have determined that selection of a parallel processor technology is an important part of the Phase II implementation effort. To this end, we have had serious discussions with three centers containing parallel processing facilities. All three would like to see us use their facilities to do this work. Selection of a facility is addressed under our proposed plan for Phase II, but final answers will not be necessary until well into Phase II when we must select a final specific implementation approach. In Phase II, we are proposing to build and test our design approach using actual runs for experimentation, leading to a final product for Phase III.

# 4. OVERVIEW OF THE PROPOSED TECHNOLOGY

## MOTIVATION FOR A NEW TECHNOLOGY

The underlying condition that allows parallel operation of the elements of a system is that they must be able to operate concurrently in an instant of time. This implies that they operate on independent information. If they are sequentially dependent, such as a sequence of equations where the variables of each successive equation depend upon solution of the prior one, then they are *not* candidates for parallel operation. If they are totally independent over some period of time, i.e., they perform their operations based upon independent pieces or copies of information, then they can be processed in parallel.

Most systems are somewhere in between. However, depending on how one designs the architecture of these systems, one may inadvertently inhibit the ability to take advantage of any parallelism when they run on a parallel machine. When developing systems that are inherently parallel, one wants to take full advantage of their parallelism. The visual architectural technology of VSE and GSS takes advantage of the parallel nature of such systems. This technology is based upon a number of new paradigms that form its foundation. We will briefly articulate these underlying concepts in this section.

## DISCRETE EVENT SYSTEMS AND SIMULATION

First, the concept of *discrete event* simulation allows modelers to more directly represent the systems they are modeling on a digital computer using a rule based (as well as mathematical) approach. Discrete event models provide the most general type of mathematical representation. Standard mathematical approaches tend to be much more abstract and inherently interconnected, resulting in less independence of the parts of the model. This is observed when trying to identify the independent parts of a real system represented in a model that closely couples them in a set of interrelated mathematical relations. This is apparent in particle physics, where many particles scatter concurrently - independently - after a few collisions. The mathematical representation of evolving dynamics may be succinct, but the independence, and therefore parallelism, is likely removed. When represented in a discrete event framework, the independence remains intact.

Next is the ability to schedule sequences of unfolding events in *discrete instances of time* in the future, based upon the state of a system or simulation at the current time. The GSS discrete event simulation approach is independent of a fixed clock time or time step, and supports a high degree of independence of models that represent independent physical entities. In GSS, *processes* that contain the rules (instructions) to be followed can be scheduled to run at any time, possibly at the same instant of time, as would their counterparts in the physical systems they represent. If they are independent, they can run concurrently. Their dependence is defined solely by the information (data) they access, contained in *resources*.

Processes that share no information directly are independent, and can be scheduled to run concurrently. In PSI's larger simulations, thousands of processes may be scheduled to run at the same instant of time. Of these, hundreds are independent, and can run concurrently. Finally, the GSS run-time monitor has all the information needed to determine the best sequence for allocating parallel processors to these independent processes to minimize running time.

This same approach can be used for real-time control and communication systems using VSE. The only difference from GSS is that VSE is using the real-time clock, while GSS uses a simulated clock. Processes scheduled in VSE occur at real clock times, while those in GSS occur immediately with the simulation clock advancing accordingly.

## HIERARCHICAL DECOMPOSITION OF GSS MODELS AND VSE MODULES

As defined here, models of independent entities can follow the same hierarchical decomposition as the physical system. This naturally preserves their independence. With this approach, it is the model *architecture,* viewed hierarchically from the top down, that is key to taking advantage of inherent system parallelism. An example of this approach is shown in Figure 4-1. Using GSS, models are easily designed and reviewed for parallelism using hierarchical graphical descriptions. Designers have direct visibility from the top of the hierarchy to the very bottom, with no layers of abstraction to cloud the architectural details. The same holds true with VSE modules.

## THE VSE AND GSS VISUAL ARCHITECTURE ENVIRONMENT

The ability to describe computer models of complex entities operating in parallel has been difficult in the past. This is because computer programming languages are designed for sequential operations of a single processor, with special commands for invoking parallel paths. Programming languages are not amenable to describing parallel system architectures. However, it is the *architecture* that is important. And this is where the VSE and GSS approach departs from the world of programming. It provides an architecture environment that allows designers to describe their models graphically, with no concern for languages. It is the architecture that determines the *independence* of models, and this is done in a graphical environment. But it is not merely the graphical representations that provide the ability to characterize the parallelism. It is the underlying primitive graphical elements that are key to the solution of effective parallelism.

### Separation Of Data From Instructions - A New Paradigm

Why do VSE and GSS offer such an effective solution? Because the heart of the parallel processing problem is knowing what groups of instructions access what data. In GSS, resources (data structures) and processes (instructions) are totally separated. This is illustrated in Figure 4-2 which contains an expanded view of the GSS model in Drawing Level 1 of Figure 4-1. Resources (data structures) are represented by ovals, and processes (rule structures) are represented by small rectangles.

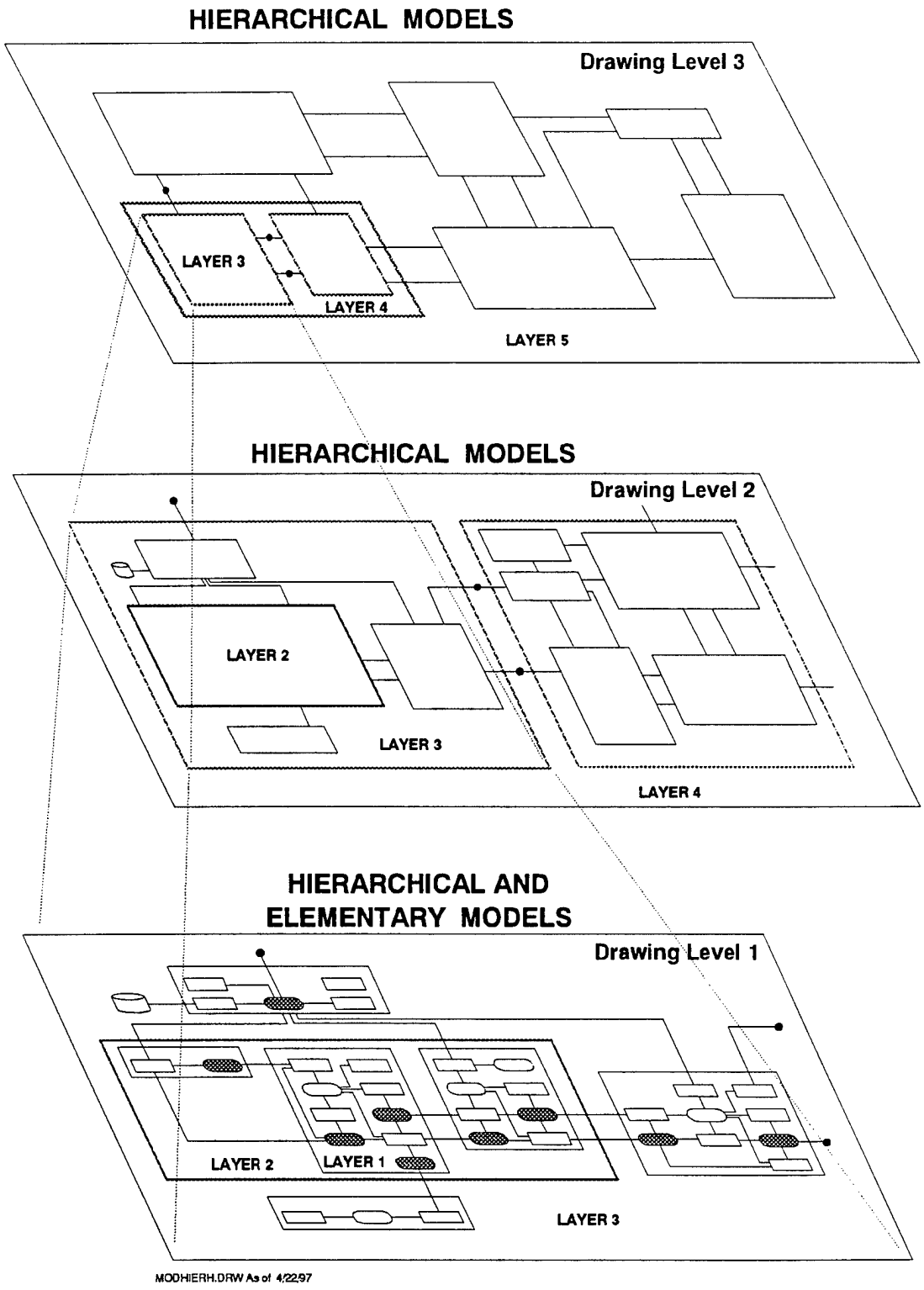


Figure 4-1. Example of a GSS (or VSE) hierarchical architecture.

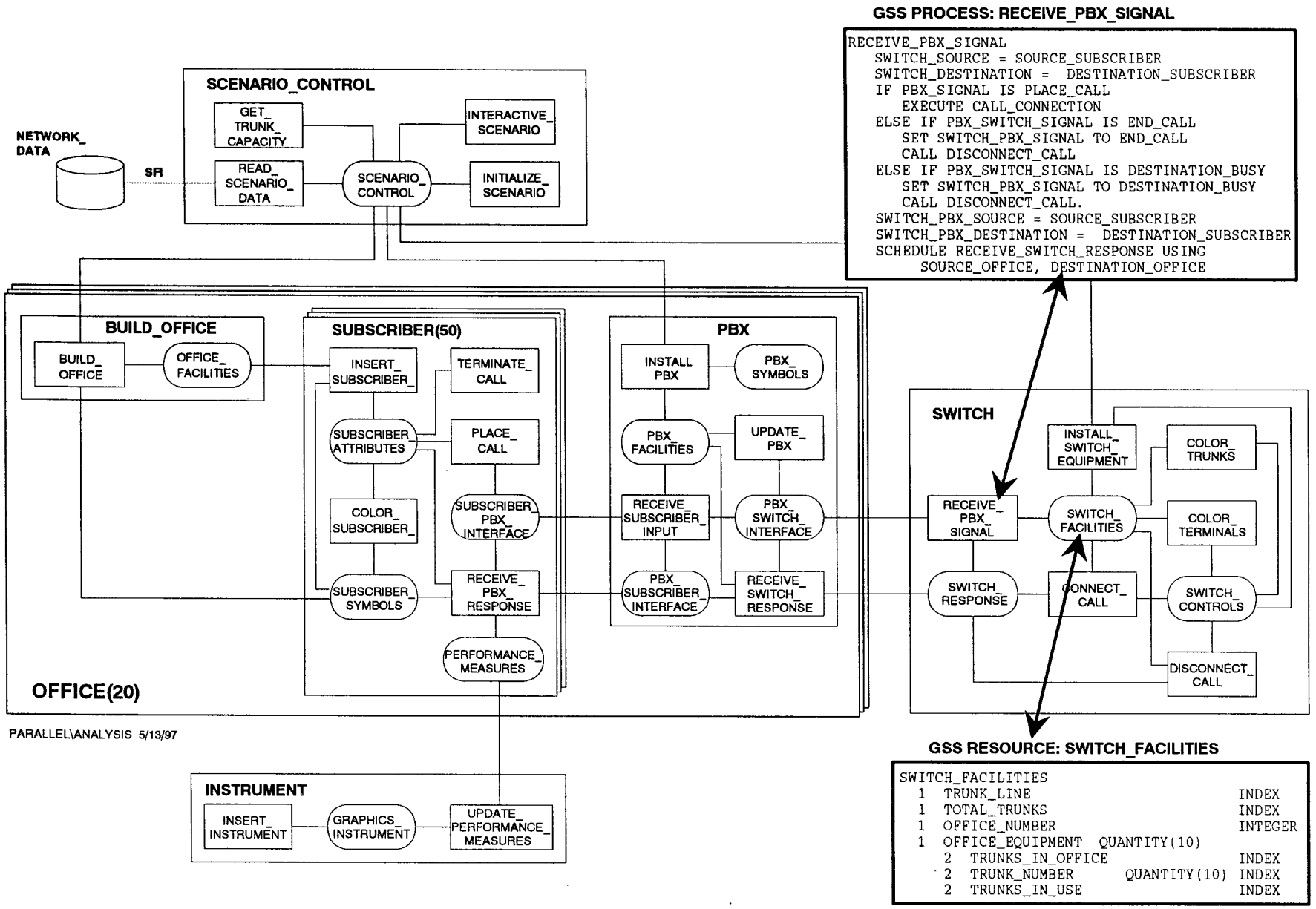


Figure 4-2. Example of a GSS Architecture.

The interconnection of processes and resources is determined solely by the graphical architecture, not in a language. The classical mixture of language abstractions that declare these interconnections, stirred in with the detailed rules governing a model, is gone! These are totally separate areas of concern, and are clearly divided in VSE and GSS. This eliminates a major area of confusion.

A model architecture is completely defined by the engineering drawing that contains information pertinent to a graphical representation. This is what supports the ability to design and visually inspect VSE modules and GSS models graphically, to maintain the parallel architecture of a system. If one follows good architectural design rules, the resulting modules will reflect independence, and therefore the inherent parallelism in a system. The design rules determine the connectivity of models that, in turn, are determined by the interconnections between them. This determination is made simply by counting the lines connecting models. By minimizing the interconnections, one maximizes imdependence of models, producing a good design.

**Separation Of Architecture And Language**

By separating data from instructions, we have succeeded in obtaining a one-to-one mapping from the drawings to the code. The visual architecture tells the total story about what processes (instructions) share what resources (data). This allows us to complete the architecture in a graphical environment, without writing code. Thus, the architecture of a GSS model or VSE module is determined independent of any language constructs. We have separated the architecture from the language, offering a visual approach to the architecture as in other engineering disciplines.

Likewise, the language descriptions are reserved for describing the data structures and rule structures of the system. There are no abstract software declarations regarding the architecture in the language, e.g., who shares what data. The language is designed to describe the detailed operation of the system.

Furthermore, since data is separated from instructions, we have focused upon convenient descriptions of data structure in a separate resource language, and convenient rule structures in a separate process language. Examples of a resource (SUBSCRIBER_ ATTRIBUTES) and a process (PLACE_CALL) are shown in Figures 4-3 and 4-4. One does not have to learn a programming language to understand the rules of a system. Instead, people who are familiar with the system's specifications can read the rules and determine if the design has been implemented properly.

We have highlighted the quantity and pointer references in the resource and process using bold typeface. This shows the classic way in which any of the 50 individual subscribers in any of the 20 offices are identified in software. In the following section, we will show how a single model can be implemented, and then instanciated as needed, with each instance being independent.

## EXAMPLE OF A GSS RESOURCE

```
05/13/97                G S S  RESOURCE REPORT                USER-ID: PSI

RESOURCE: SUBSCRIBER_ATTRIBUTES

IN MODEL: SUBSCRIBER

OFFICES_OF_SUBSCRIBERS  QUANTITY(20)

    1  PLACE_CALL_PARAMETERS  QUANTITY(50)
       2  NUMBER_OF_DESTINATIONS          INDEX
       2  DESTINATION_ID                  INDEX
       2  TELEPHONE_BOOK_ENTRY       QUANTITY(100)
          3  DEST_OFFICE                  INDEX
          3  DEST_SUBSCRIBER              INDEX
       2  CALLERS_PLAN                    STATUS PLACE_NEW_CALL
                                                 RETRY_CALL
       2  SUBSCRIBER_TYPE                 STATUS DATA
                                                 VOICE
       2  SUBSCRIBER_STATUS               STATUS BUSY
                                                 FREE

    1  CURRENT_CALL_PARAMETERS QUANTITY(50)
        2  DESTINATION_OFFICE             INDEX
        2  DESTINATION_SUBSCRIBER         INDEX
        2  CALL_TIME                      REAL
        2  CALL_DURATION                  REAL
        2  SIGNAL_TO_SUBSCRIBER           STATUS BUSY
                                                 CONNECTED
        2  PHONE_NUMBER                   STATUS UNKNOWN
                                                 FOUND

    1  CALL_ATTRIBUTES    QUANTITY(50)
       2  CALL_ORIGINATION_TIME          DECIMAL    Z(2).9(2)
       2  CALL_TERMINATION_TIME          DECIMAL    Z(2).9(2)
       2  LENGTH_OF_CALL                 DECIMAL    Z(2).9(2)
       2  CALL_STATE                     CHARACTER  22
       2  CALL_RESULT                    STATUS SUCCESSFUL_CALL
                                                 DESTINATION_BUSY
                                                 CALL_TERMINATED
                                                 BLOCKED_CALL
```

Figure 4-3. Example of a GSS (or VSE) resource.

## EXAMPLE OF A GSS PROCESS

```
05/13/97                G S S  PROCESS REPORT          USER-ID: PSI

PROCESS: PLACE_CALL     MODEL: SUBSCRIBER              TIME UNITS: MINUTES

RESOURCES:  SUBSCRIBER_ATTRIBUTES          INSTANCE POINTERS: OFFICE
            SUBSCRIBER_PBX_INTERFACE                          SUBSCRIBER
```

```
PLACE_CALL
     IF SUBSCRIBER_STATUS(OFFICE, SUBSCRIBER) IS FREE
         EXECUTE ATTEMPT_CALL
     ELSE EXECUTE RETRY_LATER.

ATTEMPT_CALL
     IF CALLERS_PLAN(OFFICE, SUBSCRIBER) IS PLACE_NEW_CALL
         SET PHONE_NUMBER(OFFICE, SUBSCRIBER) TO UNKNOWN
         EXECUTE LOOK_UP_NUMBER
            UNTIL PHONE_NUMBER(OFFICE, SUBSCRIBER) IS FOUND.
     EXECUTE MAKE_CALL

LOOK_UP_NUMBER
     DESTINATION_ID = NUMBER_OF_DESTINATIONS * RANDOM + 1
     DESTINATION_OFFICE(OFFICE, SUBSCRIBER) = DEST_OFFICE(DESTINATION_ID)
     DESTINATION_SUBSCRIBER(OFFICE, SUBSCRIBER) =

DEST_SUBSCRIBER(DESTINATION_ID)
     SET PHONE_NUMBER(OFFICE, SUBSCRIBER) TO FOUND.

MAKE_CALL
     SET SUBSCRIBER_STATUS(OFFICE, SUBSCRIBER) TO BUSY
     SET SUBSCRIBER_SIGNAL(OFFICE, SUBSCRIBER) TO PLACE_CALL
     SCHEDULE RECEIVE_SUBSCRIBER_INPUT
         USING DESTINATION_OFFICE(OFFICE, SUBSCRIBER),
               DESTINATION_SUBSCRIBER(OFFICE, SUBSCRIBER)

RETRY LATER
     SCHEDULE PLACE_CALL IN EXPON(RETRY_INTERGEN_TIME)
         USING OFFICE, SUBSCRIBER
```

Figure 4-4. Example of a GSS (or VSE) process.

## MULTIPLE INSTANCED MODELS AND MODULES

In addition to the graphical descriptions of models, GSS and VSE users can define *multiple instanced models or modules* i.e., define the number of instances of a given model, hierarchically. This is illustrated in Figure 4-1, where SUBSCRIBER(50) implies up to 50 instances within each of up to 20 office instances. PSI has designed an automated facility that provides a dramatic simplification of descriptions of instances of data structures and rule structures.

To explain this facility, the reader is referred to Figure 4-5 which shows the explicit instances of the OFFICE model. Note that the SUBSCRIBER instances are still implicit within each of the OFFICE models. However, this is sufficient to point up the independence of models inside each OFFICE instance. In particular, processes that share no resources outside the model boundaries are independent of the other models. Specifically, these processes within an instance are independent of those in other instances. This follows the physical system architecture, and allows those processes scheduled at the same time to be run concurrently.

Instanced model versions of the resource and process depicted in Figures 4-3 and 4-4 are shown in Figures 4-6 and 4-7. Note that the explicit use of subscripts is gone. This greatly simplifies the implementation. One need only specify how a model is defined for a single generic instance. Invoking multiple instances is automatic.

This facility eliminates the need for explicit pointers (subscripts) at the language level, in either the data structures or the rule structures. With this new facility, instances can be declared at the architecture level and specified when processes within an instanced model are called or scheduled to run. Otherwise, there is no need to distinguish explicitly between model instances within resources or processes. By definition, all instances follow the same rules. Their behavior depends upon their individual state vectors at a particular instant of time. Specifically, this new facility provides the following:

- The user defines the *quantity* of *model instances* and the *name* of the *model instance pointer* in the architecture environment when creating or modifying a model.
- Every resource within the model is automatically translated into multiple independent instances (copies), one for each model instance.
- *Hierarchical instances* are defined by declaring the different model instances at corresponding layers of the model hierarchy.

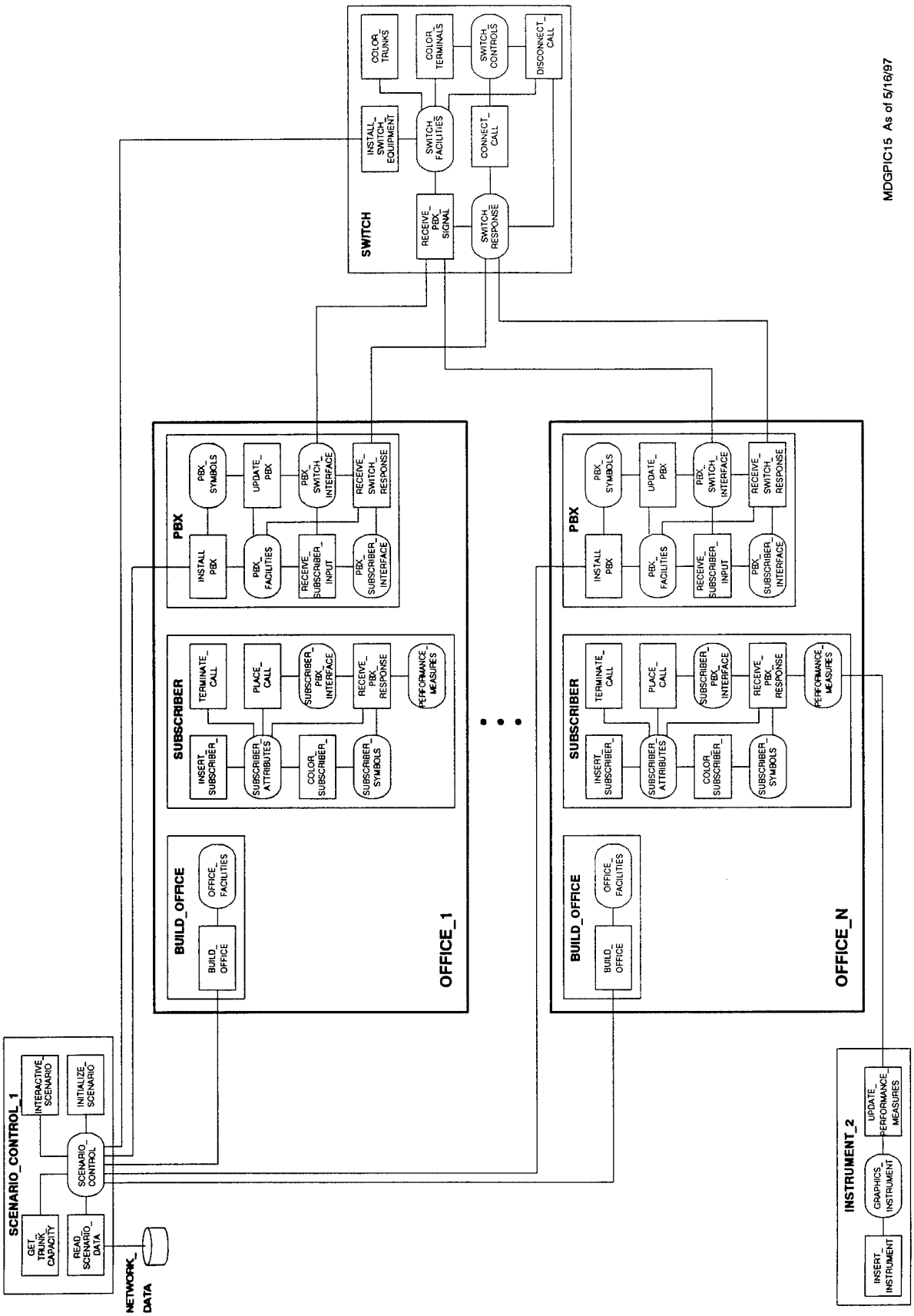


Figure 4-5. Multiple instanced office model

## EXAMPLE OF A RESOURCE WITHIN AN INSTANCED MODEL

```
05/13/97                    G S S  RESOURCE REPORT                  USER-ID: PSI

RESOURCE: SUBSCRIBER_ATTRIBUTES                                     INSTANCED

IN MODEL(INSTANCES):     OFFICE(OFFICE)
                             SUBSCRIBER(SUBSCRIBER)
```

```
OFFICES_OF_SUBSCRIBERS

    1  PLACE_CALL_PARAMETERS
       2  NUMBER_OF_DESTINATIONS          INDEX
       2  DESTINATION_ID                  INDEX
       2  TELEPHONE_BOOK_ENTRY         QUANTITY(50)
          3  DEST_OFFICE                  INDEX
          3  DEST_SUBSCRIBER              INDEX
       2  CALLERS_PLAN                    STATUS PLACE_NEW_CALL
                                                 RETRY_CALL
       2  SUBSCRIBER_TYPE                 STATUS DATA
                                                 VOICE
       2  SUBSCRIBER_STATUS               STATUS BUSY
                                                 FREE

    1  CURRENT_CALL_PARAMETERS
        2  DESTINATION_OFFICE             INDEX
        2  DESTINATION_SUBSCRIBER         INDEX
        2  CALL_TIME                      REAL
        2  CALL_DURATION                  REAL
        2  SIGNAL_TO_SUBSCRIBER           STATUS BUSY
                                                 CONNECTED
        2  PHONE_NUMBER                   STATUS UNKNOWN
                                                 FOUND

    1  CALL_ATTRIBUTES
       2  CALL_ORIGINATION_TIME          DECIMAL    Z(2).9(2)
       2  CALL_TERMINATION_TIME          DECIMAL    Z(2).9(2)
       2  LENGTH_OF_CALL                 DECIMAL    Z(2).9(2)
       2  CALL_STATE                     CHARACTER  22
       2  CALL_RESULT                    STATUS SUCCESSFUL_CALL
                                                 DESTINATION_BUSY
                                                 CALL_TERMINATED
                                                 BLOCKED_CALL
```

Figure 4-6. Example of an instanced GSS (or VSE) resource.

## EXAMPLE OF A PROCESS WITHIN A HIERARCHICAL INSTANCED MODEL

```
05/13/97                G S S  PROCESS REPORT          USER-ID: PSI

PROCESS: PLACE_CALL     MODEL: SUBSCRIBER              TIME UNITS: MINUTES

RESOURCES:                        MODEL(INSTANCE):
      SUBSCRIBER_ATTRIBUTES             OFFICE(OFFICE)
                                            SUBSCRIBER(SUBSCRIBER)
      SUBSCRIBER_PBX_INTERFACE          OFFICE(OFFICE)
                                            SUBSCRIBER(SUBSCRIBER)
```

```
PLACE_CALL
     IF SUBSCRIBER_STATUS IS FREE
         EXECUTE ATTEMPT_CALL
     ELSE EXECUTE RETRY_LATER.

ATTEMPT_CALL
     IF CALLERS_PLAN IS PLACE_NEW_CALL
         SET PHONE_NUMBER TO UNKNOWN
         EXECUTE LOOK_UP_NUMBER
            UNTIL PHONE_NUMBER IS FOUND.
     EXECUTE MAKE_CALL

LOOK_UP_NUMBER
     DESTINATION_ID = NUMBER_OF_DESTINATIONS * RANDOM + 1
     DESTINATION_OFFICE = DEST_OFFICE(DESTINATION_ID)
     DESTINATION_SUBSCRIBER = DEST_SUBSCRIBER(DESTINATION_ID)
     SET PHONE_NUMBER TO FOUND.

MAKE_CALL
     SET SUBSCRIBER_STATUS TO BUSY
     SET SUBSCRIBER_SIGNAL TO PLACE_CALL
     SCHEDULE RECEIVE_SUBSCRIBER_INPUT
         USING DESTINATION_OFFICE,
               DESTINATION_SUBSCRIBER

RETRY LATER
     SCHEDULE PLACE_CALL IN EXPON
         USING OFFICE, SUBSCRIBER
```

Figure 4-7. Example of an instanced GSS (or VSE) process.

## AVOIDING THE USE OF ABSTRACT UTILITIES

Abstractions are useful when building models of complex systems. One could not perform computer simulation without abstracting reality into models. GSS provides for ease of abstraction where complex processes, that may be spread across all of the entities in a system, are represented in a single *utility* model with a common data structure. GSS also contains high speed list management facilities that eliminate the need for the modeller to develop linked list software, a basic abstraction in modeling. However, the trade-off here must consider ease of development as well as speed and memory utilization at run-time.

With today's computers, memory availability is not an issue. The trade is usually between development time and running time, given budget constraints in dollars. This leads to decisions on how instances are represented. This choice is usually between an abstract list utility versus copies of data in independent model instances.

When using parallel processors, it is often the case that the solution that minimizes *both* development time and running time is to put the data into independent model instances. This would be considered a significant waste of memory among programmers. But with memory being abundant in a parallel processing machine, this approach provides the most significant benefits in terms of simplicity of design, development and support time, and running time on a parallel machine.

# 5. PHASE I ANALYSIS - DEFINITIONS

To support the project objectives, an analysis was performed during Phase I, supported by experimentation using PSI's existing software and simulation database to gather statistical data. This data was used to determine the best approach to dynamic processor allocation to make effective use of parallel processors. Dynamic allocation of processors can be performed adaptively during run time using automatically generated code tailored to the particular modules in a given system. This analysis was aimed at determining the rules for automatic allocation of processors to processes, including dynamic load balancing, during run-time. This analysis required agreement on the definitions below. Although GSS is used as the example here, the VSE software equivalent follows directly. *Simulation models* become *software modules,* and the *simulation clock* is the *real-time clock.*

## CLASSES OF MODELS (MODULES) AND THEIR ELEMENTS

- INSTANCED MODELS - Models can be defined to have multiple *instances* at the architectural level. This implies that, at run time, *each instance* of the model must have an *independent* copy of every resource in the model, corresponding to *instanced resources.* When invoked at run time, processes contained in an instanced model are assigned instance numbers that reference their corresponding resource instances.

- INSTANCED RESOURCES - Resources are defined to have multiple *instances* when they are elements of an instanced model at the architectural level. This implies that, at run time, *each instance* of that resource exists as an *independent copy* and is referenced by a unique name determined automatically from the resource name and instance number.

- HIERARCHICAL INSTANCED MODELS - Instanced models may be defined within instanced models hierarchically. Resources contained in the lowest level instanced model will have as many independent copies as the product of the successive instances in the hierarchy. These will be referenced by a unique name determined from the resource name and successive instance numbers. When invoked at run time, processes contained inside the lowest level instanced model are assigned instance numbers that reference the corresponding hierarchy of resource instances.

- SHARED INTERFACES - Models that are connected by shared resources have *shared interfaces.* For example, two instances of the same model have a shared interface if a process inside one instance shares a resource with a process inside another instance. The shared resource is the shared interface.

- INTERIOR AND INTERFACE ELEMENTS - Processes (resources) are *interior* to an elementary model if they have no shared interfaces with resources (processes) outside that model. They are *interface* elements if they do have such shared interfaces. The interior elements of an elementary model are interior to any higher level model containing the elementary model.

- INTERIOR AND INTERFACE MODELS - Models are *interior* to a hierarchical model if they contain no elements with shared interfaces outside that hierarchical model. They are *interface* models of that hierarchical model if they do have elements with such shared interfaces. Models that are interior at a given level of a hierarchy are interior to all higher levels.

## INDEPENDENT MODULE INSTANCES

Well engineered physical systems are modular. It is the way designers deal with complexity. Physical system modules are generally designed so their internal processes are independent of those outside the module, with the interface between modules well defined and simple as possible. Identical module copies may appear many times in the same system, interacting through interface devices or media. We refer to these copies as module *instances.* It is the concurrent operation of many such modules that make a system inherently parallel in its operation. Modules may occur as many instances of the same type, or as totally dissimilar entities. Figure 5-1 illustrates a radio system with multiple instances of the same radio sharing the electromagnetic environment. There are a number of ways to implement this model. Figure 5-2 illustrates a GSS model of this system. The question is: What is the best solution for the effective use of parallel processors as well as for the user to represent this inherently parallel system.

As described in Section 4, one of the main goals of PSI's proposed parallel software technology is to eliminate the requirement for explicit instance pointers in process statements to identify instances of a resource when referring to its attributes. These pointers are redundant, referring to model instances that can be invoked by schedule, cancel or call statements that reference the architecture level. Figures 5-3 and 5-4 illustrate this concept for a single instanced model interfaced to a double instanced model as shown in Figure 5-2. To take advantage of this facility, the following must be true.

- Model instances must have no *direct shared interfaces,* i.e., no instance can contain a process that shares a resource in another instance of the same model. Direct shared interfaces are generally considered nonphysical since real module instances are independent and interact physically through another medium.

- When scheduling a process in an instanced model, the instance must be identified. This must be explicit when scheduling from outside the instanced model, but is implicit when scheduling a process from within the same instance of the model.

- When scheduling or calling a process, the instance number of the scheduling/calling process is passed automatically to the scheduled/called process.

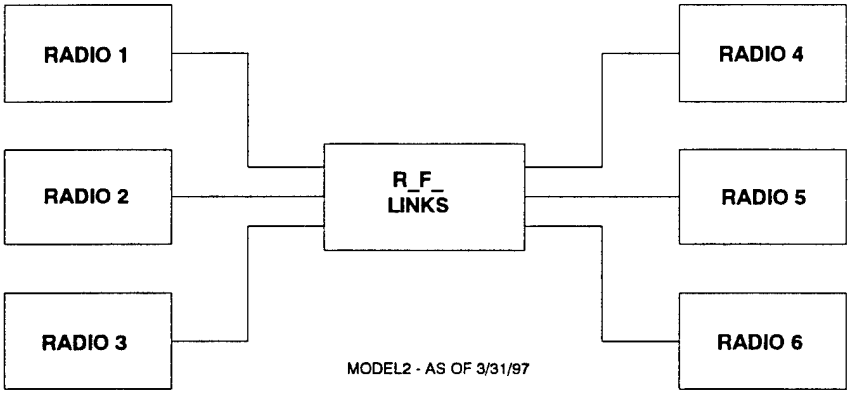


Figure 5-1. Example of interconnections of a radio system.

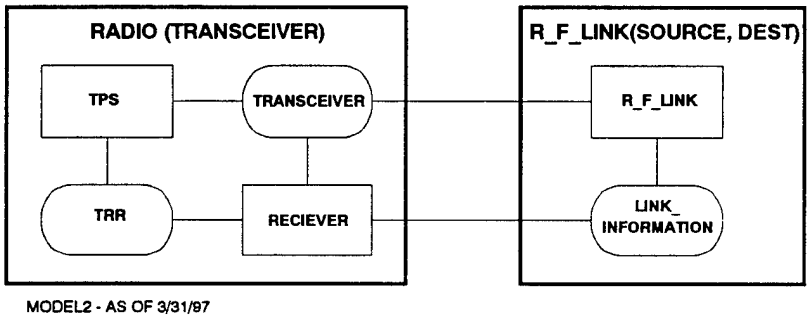


Figure 5-2. Example of interconnections of instanced models.

```
05/13/97                G S S  RESOURCE REPORT                USER-ID: PSI

RESOURCE: TRANSCEIVER                                INSTANCED

IN MODEL(INSTANCES):    RECEIVER(RECEIVER)

-------------------------------------------------------------------------

GENERAL_PARAMETERS
      1     TRANSMITTER_POWER                        REAL  INITIAL_VALUE 100
      1     RECEIVER_THRESHOLD                       REAL

TRANSCEIVER_STATE
      1     TRANSCEIVER                              STATUS      TRANSMITTING
                                                                 RECEIVING
                                                                 IDLE
                                                                 OFF
      1     TRANSCEIVER_LOCATION
            2     X_POSITION                         REAL
            2     Y_POSITION                         REAL
            2     ELEVATION                          REAL
      1     ANTENNA_HEIGHT                           REAL
      1     ANTENNA_GAIN                             REAL

CONNECTIVITY_DATA
      1     TOTAL_SIGNAL_POWER                       REAL
      1     LINK_NUMBER                              INDEX
      1     TOTAL_NOISE_POWER                        REAL
      1     LINK_CONNECTIVITY_VECTOR  QUANTITY(500)
            2     PROPAGATION_LOSSES
                  3     TERRAIN_LOSS                 REAL
                  3     FOLIAGE_LOSS                 REAL
                  3     TOTAL_LOSS                   REAL
            2     SIGNAL_TO_NOISE_RATIO              REAL
            2     LINK_DELAY                         REAL
            2     LINK                               STATUS      GOOD
                                                                 FAIR
                                                                 POOR

TRANSCEIVER_RULES
      1     TRANSCEIVER_PROCESS                      RULES TRANSMISSION
                                                           RECEPTION

      TURN_ON_TRANSCEIVER

      TURN_OFF_TRANSCEIVER

PAREXAMPLS  As of 7/30/93
```

Figure 5-3. Example of resource within a single instanced model that interfaces with a double instanced model.

```
PROCESS: RECEPTION      MODEL: RECEIVER                TIME UNITS: MINUTES

RESOURCES:                      MODEL(INSTANCE):
      TRANSCEIVER                     RECEIVER(TRANSCEIVER)
      TRR                             RECEIVER(TRANSCEIVER)
```

```
START_RECEPTION
      IF TRANSCEIVER IS IDLE
            EXECUTE VALID_RECEIVE
      ELSE IF TRANSCEIVER IS RECEIVING
            EXECUTE CONFLICTING_RECEPTION
      ELSE IF TRANSCEIVER IS TRANSMITTING
            EXECUTE CONFLICTING_BROADCAST.

VALID_RECEIVE
      IF SIGNAL_TO_NOISE_RATIO
       IS GREATER THAN RECEIVER_THRESHOLD
        AND SYNC_CODE IS A VALID_CODE
            SET TRANSCEIVER TO RECEIVING
            POWER_AT_RECEIVER = SIGNAL_POWER.
      IF MESSAGE_TYPE IS FORMAT_A
        AND LAST_SYMBOL IS A TERMINATOR
            EXECUTE SEND_ACKNOWLEDGEMENT.
      CALL DECODE_MESSAGE                    *** Within the same instance of the model

CONFLICTING_RECEPTION
      IF SIGNAL_POWER
        IS GREATER THAN POWER_AT_RECEIVER
            SCHEDULE ABORT_RECEIVE NOW.      *** Within the same instance of the model

CONFLICTING_BROADCAST
      CANCEL END_RECEIVE NOW
      SCHEDULE START_RECEIVE IN EXPON(0.83) MILLISECONDS
            WITH PRIORITY 80                 *** Within the same instance of the model

SEND_ACKNOWLEDGMENT
      SOURCE = TRANSCEIVER
      DEST   = TRANSMITTER
      MOVE ACKNOWLEDGEMNT TO TRANSMIT_MESSAGE_BUFFER
      IF DESTINATION IS BROADCAST
            SEARCH LINK_CONNECTIVITY_VECTOR ON RCVR
                  EXECUTING TRANSMISSION
                        WHEN LINK IS GOOD
      ELSE EXECUTE TRANSMISSION.

TRANSMISSION
      SCHEDULE LINK_INPUT WITH INSTANCE SOURCE, DEST
          IN LINK_DELAY(TRANSCEIVER) MICROSECONDS    *** Across different instances

PAREXAMPLS As of 4/9/97
```

Figure 5-4. Process within single instanced model that interfaces with double instanced model.

## CONCURRENT PROCESSING OF INSTANCES

In a parallel processing environment, the instances of a model CAN reside in different processors from noninstanced models that interface with it. In addition, different instances of the same model can reside in different processors. To run concurrently, processes in model instances must not have direct shared interfaces. For example, referring back to Figure 5-2, whenever one instance of the RADIO model must access link information from another instance of the same RADIO model, it must share at least one resource in the R_F_LINK model with the other instance of the RADIO model. To have a process in one instance access resources inside another instance would be a nonphysical approach. Resources that are internal to one model instance are external to all other instances. Only resources external to a model can be shared across instances.

When the R_F_LINK process is invoked, it must have access to the particular instance of the TRANSCEIVER resource it interfaces with as shown in Figure 5-2. This requires that an instanced model be interfaced with one that may or may not be instanced. In the case that process TPS in the RADIO model schedules process R_F_LINK, then R_F_LINKmust know the instance of mutually shared resource TRANSCEIVER , and this is provided automatically by GSS. Likewise, if process R_F_LINK schedules process RECEPTION, then if the R_F_LINK model (and thus LR) is instanced, the instance pointers must be automatically passed to RECEPTION.

We have defined general rules for determining the independence of processes. These rules apply to both instanced and noninstanced models (modules). We have also defined requirements for specifying pointers, and the rules for passing these pointers automatically between hierarchically instanced models.

# 6. PHASE I ANALYSIS - MEASURES OF MERIT

## MEASURES OF SPEED IMPROVEMENT

Meaningful measures of merit are required to analyze and evaluate the benefits of using parallel processor computers. Specifically, measures should include cost as well as time factors. For example, a measure of *effectiveness* for a machine with a large number of processors can be the ratio (S·M)/(C·N). S is the cost savings per unit time, e.g., how much money is saved per hour or per day if a job is completed in some fraction of the time it would take on a single processor. M is the speed multiplier, i.e., how much faster is the job done. C is the cost per processor - there is a potential increase in cost for each additional processor used; and N is the number of processors. Figure 6-1 shows a set of possible curves for the speed multiplier, M, a measure of the increase in speed (1/decrease in time) to run the same simulation using N processors versus a single processor. Curve I represents the ideal case; N processors yield a speed multiplier of N. Note that, in case C, it can decrease as more processors are used. This is due to the increased overhead encountered when trying to use a large number of processors on a problem with less inherent parallelism. These curves can be generated by running a sufficient number of experiments and varying the number of processors used for a given approach to improving efficiency.

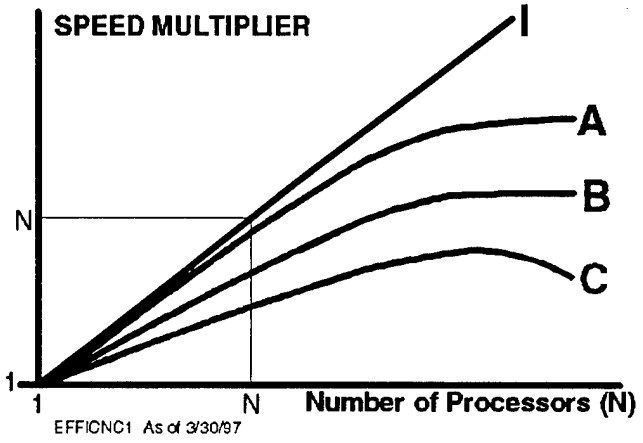


Figure 6-1. Speed multiplier versus number of processors.

We can also generate a measure of the percent processor utilization by sampling the number of processors actually working on the test problem versus those assigned. These samples can then be ordered in terms of percent processor utilization. Five different examples are shown in Figure 6-2. The area under each curve represents a measure of the percent productivity of the allocated processors for that ensemble of samples in the *simulated* time interval [T, T+ΔT]. Depending upon the work load distribution across processors, and the ordering of processor allocations to scheduled processes, these curves will change from time interval to time interval.

For the examples in Figure 6-2, curve A shows the most productivity and E the least. Note that maximum theoretical productivity over the time interval would require 100% productive processor utilization for the entire interval, allowing no time for overhead functions. We believe that a 90% processor productivity can be achieved with a large number of processors on an EPLRS type simulation using the technology proposed here.

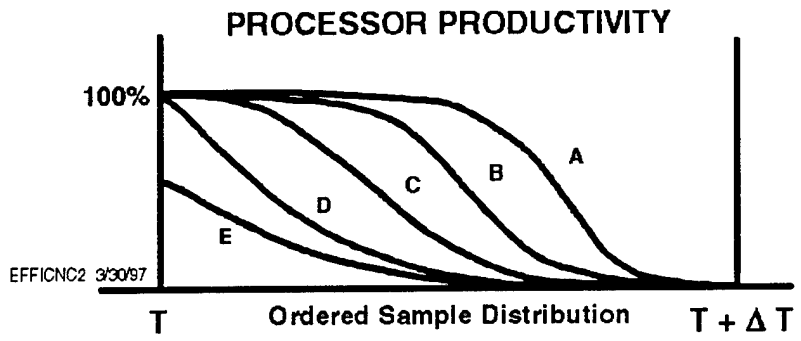


Figure 6-2. Ordered sample distributions of processor productivity in (T, T+ΔT).

In looking at Figure 6-2, one can conceive of ways to take advantage of inherent parallelism in the software system itself, accounting for independence. For example, processor productivity during any time interval will depend upon the order of processes presented for processing at that time. Candidate processes for concurrent processing must be independent from those currently running, or they cannot be started. However, ordering the start times can affect processor productivity. For example, if the order of processes presented causes blockage, then processes must wait. This occurs when two processes that are independent of each other share a resource with a third. If the two independent processes run concurrently, then the overall time to run is shortened compared to having one run before the third and the other run after.

If a number of processes are candidates for concurrent processing, the order in which they are run can clearly affect the real-time spent running them. In Figure 6-2, ΔT is simulated time. However, the amount of real time required to process the order in curve E must be much longer than that required to process curve A.

Clearly, the ability to assign processors to processes depends upon the statistics of a given simulation. It also depends upon the scenario, and even the different stages in a given scenario. We must tailor our design to these statistics, and consider providing adaptive mechanisms that will work differently for different situations. This means that we must design a run-time environment that can track the statistics as they change, and adjust accordingly. Types of statistics include numbers of processes that are scheduled at the same time and priority, and the balance of processor assignments.

# 7. DESIGN OF EXPERIMENTS TO SUPPORT ANALYSIS

In Phase I, PSI designed and performed experiments to collect statistical data to support analysis of a number of implementation concepts that we had previously postulated for allocating parallel processor resources effectively. Over the past eight years, we have analyzed our own large scale simulation and software database to select candidates for instrumentation. We concluded that simulation represents a more difficult application to accommodate than real-time control. This is because the simulation clock cannot be advanced until all processes have completed at their scheduled time. Otherwise, time would have to be turned backward, a nonlinear problem, reference lessons learned from the Time-Warped Operating System, Rieher [3]. This places more stringent requirements on memory coherence. In real-time control and communication systems, coherency specifications are met by proper hardware selection and software design. During the Phase I effort we reaffirmed our conclusions that any solutions for simulation should accommodate real-time control and communication systems.

As part of this analysis, we instrumented the second largest of PSI's simulations, the Multi-Switch Simulation (MSS), and ran various experiments to determine the feasibility of generating the special code required to take effective advantage of hundreds of processors. This selection was based upon a number of factors. First, MSS has an extremely high degree of inherent parallelism. We could likely use on the order of 1000 processors under large scenarios. Second, we can easily vary the inherent parallelism simply by changing the scenario. We cannot assume that there will always be more processors available than processes to run in parallel. Such an assumption would not constrain us to priority allocation, a more realistic situation that we believe represents our client's real-time control and communication applications. Our own analysis shows highly scaleable Shared Memory Programming (SMP) architectures to be most suited to supporting these requirements. By "shared" memory, we imply that memory segments available to one processor can be made available to another in a manner that supports expected coherence of use.

The JIEO version of Multi-Switch Simulation (MSS) generates approximately 450,000 lines of C code automatically. This same simulation was also used as a test bed for the Phase I experiments. We instrumented this simulation using our own Silicon Graphics INDY computer.

At the heart of the MSS are detailed models of the Army's MSE Circuit, Packet, and ATM switches produced by GTE and BBN. Multiple instances of any switch type can be grouped into a network, and multiple networks can be formed using gateways between the circuit and ATM networks or using a special piggyback configuration developed by GTE for using the ATM network as a backbone for packet switched networks. Each switch model is extremely detailed, with up to 500 instances in a given scenario. Scenarios include dynamic voice calls and data sessions being generated from any of thousands of subscribers. This simulation is currently running on an IBM RISC-6000 and an SGI INDY with real-time to simulation-time ratios running from less than 1:1 to over 100:1, depending upon the scenario. We have run a number of large scenarios to test ways to make use of a large number of parallel processors to cut the running times of this simulation. For example, an improvement of 100:1 would allow 5 hour runs to be cut to 3 minutes.

## APPROACH TO STATISTICAL DATA COLLECTION

To collect data that would provide the basis for evaluating inherent parallelism and corresponding opportunities for concurrent processing, we set down criteria for concurrent processing, i.e., what are the properties of a process that makes it a candidate for parallel processing. We then determined how often these properties occur and how they relate to the architecture, particularly the graphical connection. We collected sufficient data to determine the feasibility of parallel processing, and estimated the potential efficiencies of processor utilization.

## CHARACTERIZING THE SCHEDULER QUEUE

Opportunities for concurrent processing are generally determined by the contents of the scheduler queue. An illustration of the scheduler queue is shown in Figure 7-1. This queue contains an entry for every process scheduled to be run. As shown in the illustration, many processes can be scheduled at the same time and with the same priority. These are candidates for running concurrently. For very large simulations (systems), particularly those using large numbers of instances of a given model (module), the number of processes scheduled concurrently (same time and priority) can range into the thousands.

Before one can determine the best way to allocate processors to processes, one must understand fully the stochastic nature of this queue. All processes scheduled at the same time with the same priority are candidates for running concurrently. Therefore, one can scan the queue to determine which processes are candidates for concurrent processing. Also, processes that are running can schedule other processes to run NOW (i.e., at the current clock time). Therefore, a process that is currently running can schedule one that can start running concurrently, immediately, implying that one does not know the full picture of opportunities for parallel processing at the current time until the clock advances to the next time step. Statistics must be gathered to determine the percentage of outcomes in this category.

Finally, one must consider that the scheduler itself can be running concurrently in a separate processor. This would allow a process to be run concurrently with the process that just scheduled it NOW, provided they were independent - an unlikely case since they most likely share a resource.

Statistics had to be gathered in a manner that provides an accurate picture of scheduler dynamics that can be used to characterize the stochastic properties of the queue for representative scenarios. The approach insured capturing snapshots of the queue in a manner that exposed the different cases that can occur as described in the previous paragraphs.

Having characterized the scheduler queue, we must take a deeper look into the mechanics of concurrent processing, and the rules that can be used to determine if a process is a candidate for parallel processing.

| TIME | PRIORITY | PROCESS NAME | INST-1 | INST-2 | INST-N |
|---|---|---|---|---|---|
| 10.000 | 50 | PROCESS-AA | | | |
| 10.000 | 50 | PROCESS-CC | 3 | | |
| 10.000 | 50 | PROCESS-DD | | | |
| 48.000 | 20 | PROCESS-FF | | | |
| 48.000 | 50 | PROCESS-GG | | | |
| 100.000 | 50 | PROCESS-HH | 1 | 1 | |
| 100.000 | 50 | PROCESS-HH | 1 | 2 | |
| 100.000 | 50 | PROCESS-HH | 1 | 3 | |
| 100.000 | 50 | PROCESS-II | 2 | 1 | |
| 100.000 | 50 | PROCESS-II | 2 | 2 | |
| 100.000 | 50 | PROCESS-II | 2 | 3 | |
| 119.000 | 50 | PROCESS-KK | | | |
| 250.000 | 50 | PROCESS-MM | 9 | | |
| 280.000 | 50 | PROCESS-SS | 7 | 1 | |
| 280.000 | 50 | PROCESS-SS | 7 | 2 | |
| 280.000 | 50 | PROCESS-SS | 7 | 3 | |
| 280.000 | 50 | PROCESS-SS | 8 | 1 | |
| 280.000 | 50 | PROCESS-SS | 8 | 2 | |
| . | . | . | . | . | |
| . | . | . | . | . | |
| . | . | . | . | . | |

Figure 7-1. The GSS scheduler queue.

## ANALYSIS OF PARALLEL SCHEDULING RULES

As defined previously, two processes are independent if they do not share any resources. We now define *connectivity* as the inverse of independence, such that two processes are connected if they share a resource. Thus processes are independent if and only if they are not connected. In VSE or GSS, the connectivity of a module is defined graphically in the architecture environment, not the language environment. The VSE and GSS top level monitors maintain this information. It is available in the form of a connectivity matrix during run-time.

To determine if a process can be run concurrently on a processor, one must determine if that process is independent of those currently running. This requires the following checks.

- To be a candidate for concurrent processing a process must be scheduled to run at the same time as the current process and with the same priority.
- For the candidate process, we must determine if it is independent of every process that is currently running. If not, go on to the next candidate process at the current time.

Ideally, this candidate list would be ordered such that the next process in the current time list that is most independent of the others will be scheduled first. This can be evaluated using a *relative independence measure,* to determine an effective value for the schedule key. These processes would then be ordered for review based upon the value of their relative independence measure. This can be stored in the connectivity matrix, as a cross reference list of dependencies.

Processes interior to a module instance may be *called* instead of being scheduled, and are thus invoked directly from the point at which they are called. These process calls do not affect the schedule queue, and will serve to increase processor productivity. Additional data must be gathered on these processes.

We define a *thread* as a set of processes in which the first process is scheduled and the remaining processes are *called* successively. We note that a thread is run most efficiently on a single processor.

There are additional mechanics of this environment to be characterized, e.g., the nature of the dynamic changes to the schedule during the time slice versus the state at time T. This will affect algorithm design for optimal ordering in minimal time.

Instanced modules create special submatrices of the connectivity matrix that are themselves independent. These become candidates for quasi-independent queue management, potentially in separate processors. In addition, processor load balancing must be considered, and this has been the subject of much prior research, see for example Pargus [10]. We have developed a scheme for migrating processes, on a model-by-model or instance-by-instance basis.

We will also cover the possibility of scheduling processes ahead in time by a fixed $\Delta T$, and the corresponding trade-off between speed and accuracy. PSI has performed a prior analysis of a real-time simulation split across two machines, each with their own simulation clock synchronized to within a prescribed $\Delta T$. As $\Delta T$ is increased, more processes become candidates for concurrent processing, still having to meet the independence criteria. As $\Delta T$ is increased from 0, changes in simulation output can be characterized, and an error function created as a function of $\Delta T$. One can then decide upon speed accuracy trade-offs. This concept is considerably different from that of the Time Warped Operating System where there are no limits on $\Delta T$ and sequences of processes improperly scheduled in advance must be recalled and reprocessed over, see Rieher [3].

To achieve our goals of characterizing the statistics governing this environment, we instrumented our experiments on PSI's SGI INDY computer. Without a very large and inherently parallel piece of software such as the MSS, distortions in memory and processor utilization could occur. This is because the application system itself demands significant processor and memory resources to run, and will compete heavily with overhead functions. We therefore took data to determine the various model, submodel, and process instance running times, and the opportunities to eliminate instruction loading and data movement (paging) when knowledge of the model topology is already stored in the connectivity matrix. This required the development of a statistical map of process scheduling against processor utilization. This has allowed us to postulate optimal design approaches to allocation of processes to processors, optimal ordering of the allocation process, and selection and migration of module instances to achieve dynamic load balancing.

# 8. REVIEW OF FINDINGS

As part of the Phase I effort, PSI instrumented the Multi-Switch Simulation (MSS) and the MIL-STD-188-220 simulation and ran representative scenarios. These runs produced results that were surprising at first but, upon reflection, appear obvious. To put these results into perspective, we will first describe the EPLRS Simulation Facility (ESF), the largest simulation we are aware of, and one that is clearly a candidate for parallel processing. We will also characterize properties of simulations that can be used to determine a system's eligibility for effective use of parallel processors. This generalization represents a change in our thinking regarding the application of parallel processing to more general problems.

## EPLRS SIMULATION FACILITY (ESF) - ANALYSIS OF RESULTS

The EPLRS Simulation Facility (ESF) contains two simulations: the EPLRS Connectivity simulation and the EPLRS Capacity simulation. These two simulations are considerably different in that the connectivity simulation is principally mathematical, being used to determine the probability that a message will be received for each link in the network. The output is a sequence of matrices containing these probabilities and other pertinent link information as a function of time.

The connectivity simulation is not very large and generally runs very fast, using the Fast Propagation Prediction System (FPPS). FPPS contains special algorithms, developed by PSI for the U.S. Army, to perform fast and accurate propagation path loss calculations. There are some opportunities for parallelism in this simulation, but it runs very fast on a single processor machine and is not a good candidate for improvement using parallel processors, at least in terms of time or budgets.

The EPLRS Capacity simulation is quite different. It is the largest simulation of a communication system known to PSI. It currently takes two hours to run a four hour scenario of approximately 130 radio units. It has tremendous inherent parallelism since the system being represented itself contains many processors in each radio unit running concurrently. Its users would like to run much larger scenarios, upwards of 600 radio units, and run these simulations many times varying the starting random number seed to obtain representative distributions of key measures of performance.

We will now analyze the functional and technical properties of the EPLRS Simulation Facility.

## Functional Properties of Simulations Relative to Parallel Processing

The EPLRS Capacity simulation is clearly a *functional* candidate for gaining order of magnitude improvements using hundreds of parallel processors in that it has the functional properties necessary to make efficient use of these processors. These functional properties are listed below.

- an extremely high degree of inherent parallelism,
- a large number of model instances that are independent,
- a clear time and budget incentive to speed the running time by orders of magnitude.

## Technical Properties of Simulations Relative to Parallel Processing

To gain a perspective on the *technical* aspects of applying parallel processors to this simulation, we will consider additional properties of the EPLRS Capacity simulation. While performing certain analyses a few years ago, PSI instrumented the schedule queue for this simulation. During medium to high scenarios, this queue would contain 6,000 to 10,000 processes at any instant of real time. This led to research on the design of a new scheduler. Part of this research focused on the distribution of scheduled times and priorities.

The important result was that for the larger scenarios, the EPLRS Capacity simulation had typically 1000 to 2000 processes scheduled at the current simulation clock time. This implied that, technically, these processes were candidates for running concurrently. The final determining factor for realization is, as always, data independence. If a process that is a candidate for running concurrently shares no resources with any that are already running, then technically it can be run concurrently. A large number of processes scheduled at the same time were in separate model instances and therefore had this property.

Thus the technical properties of the EPLRS Capacity simulation are:

1) A large number of candidate processes scheduled at the same simulation time and priority.

2) A very high percentage of these processes were independent (they were generally in separate model instances).

By our current technical measures, processes can be run concurrently provided: (1) that they are scheduled at precisely the same simulation clock time with the same priority; and (2) they share no resources. The first technical property appears to be peculiar to a certain class of simulations into which the EPLRS Capacity simulation falls.

## MULTI-SWITCH SIMULATION (MSS) - ANALYSIS OF RESULTS

We anticipated that queue results with the MSS would be similar to that of the EPLRS Capacity simulation. Although it is now apparent why the test results were totally different, we were surprised at first. The MSS produced very few scheduled processes with the same clock-time in the queue at an instant of time. This would imply a very low technical candidacy for parallel processing as measured by the first technical property described above. This was surprising since the MSS has a large number of independent model instances, each with many processes performing virtually identical functions. In retrospect, there is nothing to ensure that these processes will be scheduled at precisely the same clock time.

In this regard, the EPLRS Capacity simulation is very special. All radios, and thus model instances, perform their functions on time-slot boundaries or at fixed intervals from these boundaries; and all radios are synchronized to a master clock. The system itself is totally synchronized, so there is temporal synchronization of otherwise independent model instances.

## MIL-STD-188-220 SIMULATION RESULTS

The experiments on the MIL-STD-180-220 simulation produced results similar to the MSS. Normally there was only one process scheduled at the current clock time, implying that this simulation was not a good technical candidate for parallel processing. However, we again decided that this conclusion did not make sense since there are many radios in the real system operating concurrently, and the simulation should afford highly effective use of parallel processors given the independent model instances.

## REVIEW OF THE PROPOSED TECHNICAL PROPERTIES OF SIMULATIONS

The technical properties used to classify processes in the EPLRS Simulation as suitable for parallel processing appears to invoke too strict a test for the simulations analyzed in this study. Simulations that appear to have a high degree of inherent parallelism, e.g., the MSS and the MIL-STD-188-220 simulation, do not qualify for parallel processing when processes are required to have identical schedule time and priorities. Yet processes in independent model instances represent entities that run concurrently in the real systems, and therefore should be candidates for parallel processing.

When investigating technical properties, we must ensure that they can be used directly, or extended, in a way that:

- applies to software systems in general
- are usable in algorithms that test for concurrent assignment of GSS or VSE processes (threads) to processors.

# 9. TIME DIFFERENTIAL CONCURRENT PROCESSING

The basic problem with the technical properties described in Section 8 involves two or more processes that are functionally candidates for concurrent processing but do not possess the property of being scheduled at precisely the same time with the same priority. Yet, we know that in many cases we can run these processes concurrently and produce valid simulation results. An analysis of this situation is presented below.

## REAL-TIME SIMULATORS AND TEST DRIVERS

The problem described above is common to real-time simulators and test drivers used in engineering, where live users and other hardware devices can interact with a simulation. In these cases, the simulation clock is generally paced by the real-time clock. Using GSS, this is accomplished by defining the ratio of time change of the simulation clock to that of the real-time clock. When tied 1:1 with the real-time clock in a computer, one still has to be concerned with differences between the simulation clock and the computer's real-time clock, and then between the computer's clock and the clocks in other computers or hardware devices. In general, these will all be different.

The property of concern is how far apart these clocks can drift. This is determined by a measure, $\Delta T$, representing the difference between the simulation clock and any other clock in the system that is expected to be synchronized with that clock. One can then determine the point at which the time differential is too large to insure validity of results of a field or laboratory experiment.

## VALIDATION TESTING FOR MAXIMUM CLOCK DRIFT

Although the knowledge of validation testing to determine maximum clock differentials in stimulators is not wide-spread, there is considerable experience on this subject. The key concept is understanding that the validity of complex tests of this nature requires comparing statistical distributions of a sufficient number of trials to produce a valid statistic. Any real-time test requires selecting one clock as the master to compare against. Other clocks are then characterized in terms of their time differential from the master clock. One must then deal with the *clock drift* associated with other clocks in the system or test equipment.

To accomplish this, one must run a sufficient number of experiments to produce statistics of the performance measures for a given amount of maximum clock drift, $\Delta T_{max}$, within one experiment. Then one can plot the variations in performance measures as a function of maximum clock drift. Given such plots, it remains to determine when the variations are too large to represent valid measures of performance. This, in turn, determines the maximum allowable clock drift to ensure valid results from an experiment.

## MAXIMUM SIMULATION CLOCK DIFFERENTIAL ($\Delta$Tmax)

Maximum clock drift in real-time systems corresponds to a maximum simulation clock differential ($\Delta$T) in simulations. It is a measure of the maximum difference between current simulation clock-time and the next scheduled clock-time in the scheduler queue. If one is using parallel processors to run a simulation, then each processor can have different processes running concurrently, scheduling other processes (or itself) to run. At any point in time, the next process in the overall schedule queue will be scheduled at either the current time (NOW) or some time in the future. The difference between the future time and the current time is $\Delta$T.

To understand the conditions of concern, let's consider the following scenario. A given process, A, running at current time T, may schedule another process, R, to be run at some T+$\Delta$T time in the future, where this time happens to be the next time on the queue. Subsequently, process A may continue to schedule additional processes before it terminates. One of these additional processes, S, may be scheduled NOW, implying at the current clock time, T. Then process S would be put in front of process R in the queue.

Now let's consider that we will allow processes to be scheduled concurrently on parallel processors provided that $\Delta$T does not exceed some predetermined maximum value, $\Delta$Tmax. As the $\Delta$Tmax window widens, more processes will become candidates for concurrent processing.

Consider the case where process A schedules R to be run at T+$\Delta$T, where $\Delta T < \Delta Tmax$. Then R will be allowed to run concurrently with A. A then proceeds to schedule S to run NOW.

Clearly, S can run concurrently provided it is independent, i.e., it shares no resources with A or R, or any other processes that are running concurrently. However, if R schedules another process that shares a resource with S, wherein the operation of S depends upon the contents of that resource, then the order of scheduling can affect the outcome of the remainder of the simulation and the resulting measures of performance. One must be concerned about the validity of these results. One can also envision processes in independent model instances that can be run concurrently with different clock times and still produce identical measures of performance. The question is, how big can $\Delta$Tmax be and still preserve validity.

Determination of $\Delta$Tmax is similar to determination of maximum allowable clock drift in real-time systems. One must run sufficient simulations varying the random number seed to produce statistics of the performance measures for a given $\Delta$Tmax. Then one can plot the variations in performance measures as a function of $\Delta$Tmax. Given such plots, it remains to determine when the variations are too large to represent valid measures of performance. This, in turn, determines $\Delta$Tmax for a given simulation with a given scenario. Note that, in general, the value of $\Delta$Tmax will depend upon the scenario.

## THE EFFECT OF MODEL INTERFACE PROCESSES ON ΔTmax

Based upon current model architectural design approaches at PSI, interface processes, i.e., those that share resources with other models, depend upon priority to insure coherence of the data in the common shared resources. Figure 9-1 illustrates the type of configuration of interest. Let's assume that TPS1 schedules LP1, and TPS2 schedules LP2. When process TPS1 schedules process LP1 NOW (at the current clock time), it will typically use a higher priority to insure that LP1 runs prior to TPS2 so that LR1 does not get changed until LP1 runs with the data from TPS1. This implies that the schedule times of TPS1 and TPS2 must always be kept in order, else the coherency of data in LR1 can be compromised. Likewise, LP1 and LP2 must be kept in order with TPS1 and TPS2, else the data in LR1 can be compromised. Since there can be multiple schedules of any of these processes within the ΔTmax time interval, it appears that interface processes may have to be held to run with ΔT = 0, i.e., they must be synchornized.

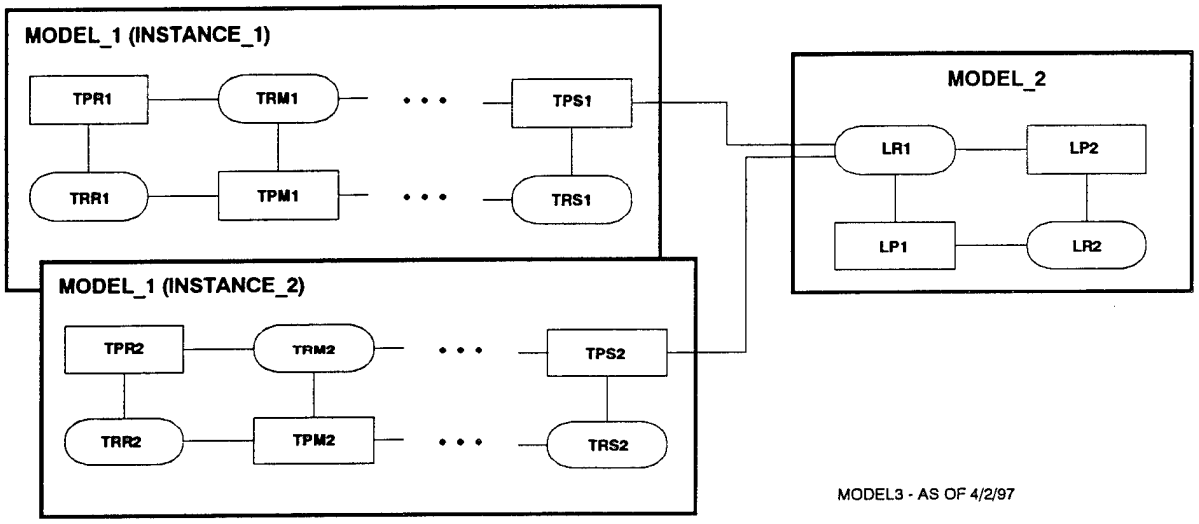


Figure 9-1. Time differentials across instanced models.

When interior process TPR1 runs, it is not updating any resources outside the model instance. If the model instance is running on a single processor, then all processes within that instance must run sequentially, in their normal order. Thus, it appears that processes interior to a model instance can run in advance, up to the ΔTmax time interval, without causing incoherence.

## THE EFFECT OF THREADS ON PROCESS SELECTION

In the above descriptions, we have always referred to scheduled processes. However, we must consider that these imply *threads*, as defined in Section 7. Threads are really what get scheduled, since any call sequences are part of the scheduled process. Therefore, when checking independence, we must check all of the resources used by every process called in the thread to see if any are currently in use.

# 10. CONCURRENT PROCESSING PROPERTIES OF SYSTEMS

We want to establish a simple set of properties with which to classify systems and simulations relative to the effective use of parallel processing. The degree to which a system possesses these properties must be indicative of how effectively one can use parallel processors to run the system. We will address simulations to start, and then extend this to software systems in general.

## PROPERTIES REQUIRED FOR CONCURRENT PROCESSING IN SIMULATIONS

The following technical properties are proposed as conditions for concurrent processing of processes in simulations. Note that "process" implies "thread" as defined in section 7.

### Independence Of Processes That Are Candidates For Concurrent Processing

This property was covered in Section 5. It is determined by the architecture of independent modules to ensure that the candidate process shares no resources with those already running. This is a necessary condition for concurrent processing.

### Number Of Processes In The Queue At The Same Time And Priority

This property was proposed and reviewed in Section 7. It is measured by taking snap-shots of the scheduler queue at representative points in time to determine the number of processes scheduled at the same time and priority. These are candidates for concurrent processing.

### Number Of Processes Scheduled At The Same Time And Priority

This may be appear to be the same as Number Of Processes In The Queue At The Same Time And Priority, but it is different. It is important when snapshots of the queue at any time show that the number of processes in the queue at the same time and priority is small compared to the number of processes that get scheduled at the same time and priority. This typically occurs when a process schedules another process NOW. This puts one additional process on the queue at the current time. On a single processor machine, the scheduled process does not run until the scheduling process completes. If the scheduled process continues the chain of scheduling another process NOW, then many processes can be scheduled at the same time and priority while only one appears in the queue at any point in time.

In GSS, processes are entered into the queue as they are scheduled. In a parallel processing environment, they could run concurrently with the scheduling process. This could cause many processes to be scheduled at the same time and priority, even though there are no more than one in the queue at any time having the same time and priority.

### Maximum Size of Discrete Clock Interval - ΔTmax

In systems where there are large independent modules, typically module instances, it may be possible to force schedule times to discrete clock intervals while producing valid simulation results. This would tend to line up process schedule times, eliminating some of the unnecessary drift that is encountered when generating random samples for schedule times. As schedule times align, more opportunities occur for concurrent processing.

In most cases, one instance does not depend on the schedule times of another. A case where this may cause an invalid result is when two independent modules are interfaced, and the schedule times at the interface must be in a given sequence for proper operation.

Another way to implement a discrete-time scale when scheduling processes is to draw discrete-time random samples when determining intergeneration times. This could be accomplished by defining a descretized distribution in terms of number of intervals, and then by translating a real-valued sample to the nearest discrete boundary.

To accommodate these facilities, it may be possible to use a discrete clock scale when scheduling processes. In that case, we can designate the level of resolution of descretization by defining the maximum clock interval size - ΔTmax - that still ensures validity of results. Determination of ΔTmax via experimentation would be done as described in Section 4.3.

### Maximum Size of Advance Schedule Interval - ΔTmax

In systems where there is no synchronization between independent modules, data is exchanged at times that are independent of the clocks in the receiving modules. It is the reason that few if any processes are scheduled at the same time and priority in the MSS and MIL-STD-188-220 simulation. This reflects real system behavior.

When model architectures are designed along the same lines as the physical modules, independent models or model instances should be further immune from some degree of "clock drift". Having one model instance run slightly ahead of another clock-wise (effectively a clock drift in a module) should not cause loss of validity until some ΔTmax clock differential is reached. However, processes lying within the same model instance that have been scheduled at slightly different times may adversely affect validity if run concurrently. It is possible that this situation can be taken care of by proper design of the run-time scheduler by allowing only processes that are interior to independent models or model instances to be run concurrently if scheduled ahead of the actual simulation clock time.

Clearly, as ΔTmax grows, more processes fall into the window for concurrent processing. The question then becomes one of finding the value of ΔTmax that ensures validity of results. Determination of ΔTmax via experimentation is described in Section 4.3 above. As indicated before, this concept is quite different from that in the Time Warp Operating System described by Rieher, [8], in that it limits advance scheduling and involves no reprocessing.

## CONCURRENT PROCESSING IN REAL-TIME SYSTEMS

Real-time systems are valid by design - assuming design criteria are met. If processes are scheduled to run at some time in the future, they are invoked when the real-time clock advances to that time and causes an interrupt. At that point, one is only concerned about fielding events in real-time to meet time requirements. Parallel processing may be required to meet such design constraints, in which case validity of results is implicit in the design.

Using parallel processors for simulation is more difficult than for real-time systems. However, the inherent parallelism in simulation is generally much higher than for real-time systems, and presents a large existing market for effective solutions.

When looking at real-time systems, one finds that many of the functions may be done in different processors. These processors may be distributed geographically. So we must consider the aspects of distributed processing, another form of parallelism.

## CONCURRENT PROCESSING IN DISTRIBUTED SYSTEMS

A distributed system is one that contains subsystems running concurrently on separate processors spread geographically. As systems become more complex and cover more functions, they appear to be heading in this direction. In a sense, they are systems of systems. From our standpoint, we must address this direction as one that apparently is becoming the mainstream of the future.

Systems may be distributed in multiple processors within a single room, within a building, within a city, or over many nations. However a system is distributed, there can be many processors running critical subsystems concurrently. This is an evolving form of parallel processing, and we must consider this type of environment as decisions are made for a visual architectural approach to software design.

One of the most widely known distributed systems in DoD is the Distributed Interactive Simulation (DIS) network environment. This provides for distributed experiments - both live and simulated - to be interconnected into one big (and hopefully more realistic) experiment. PSI has participated in these experiments for a number of years, either directly, or through its clients that are contractors to the DIS proponents. These systems can contain many different but interconnected simulations running on a network of computers. The network is used to pass data among the simulations.

In working this type of problem, the major concern is synchronization of otherwise independent simulation clocks. This problem is akin to finding $\Delta T_{max}$ for the maximum size of advance schedule intervals and corresponding window of opportunities for concurrent processing. This is because it is not necessary to wait for precise clock-time synchronization among independent subsystems to achieve validity since, by definition, it is achieved provided the independent simulation clocks do not get out of sync by more than $\Delta T_{max}$.

# 11. REVIEW OF OBJECTIVES & PROPOSED APPROACH

## REVIEW OF OBJECTIVES

Given the technical properties described above, we will now review the objectives and proposed approach to accomplish the main goal of this project, namely to make the effective use of parallel processors transparent to the software module or model designer.

To implement this approach, there are two elements required to achieve our overall project objectives. These are:

- A visual (graphical) approach to the architecture to insure taking maximum advantage of the inherent independence properties of a system and thus parallelism at run-time.
- A run-time environment that *automatically* makes effective use of parallel processors while supporting the wide variety of problem types and machine environments expected.

## REVIEW OF THE PROPOSED APPROACH

PSI has already developed a visual approach to software architecture that insures mapping the inherent parallelism of a system (or simulation) into independent modules (or models). What is now required are enhancements to the translation systems and run-time environments that will automate the mapping of independent modules or models into parallel processes.

In Chapter 10 we reviewed the properties of systems required to support parallel processing. We reviewed the requirements for simulations as well as real-time systems. We must now consider these properties as they apply to the SMP and networked (distributed) computer environments.

In the case of distributed computers, whether for real-time systems or simulations, two major design factors must be considered. One design factor is updating resources that are shared across processor boundaries. This is because copies of these resources must be kept in the separate processes. The second design factor is allowing for clock drift between machines. In this case, the most simple solution from a real-time system or simulation designer's standpoint appears to allow scheduling to occur in advance and experiment with a value of $\Delta T_{max}$ that ensures validity.

Designers of systems using parallel machines may have to consider tradeoffs when speeds cannot be achieved using the techniques proposed here. It may turn out that, no matter how the implementation is done for either systems or simulations, the required speeds cannot be achieved given the current state of hardware technology. However, based upon our experience with

simulations and distributed systems, we believe that one to two order-of-magnitude improvements can be made when functional and technical properties permit, and that there are many important cases that meet the property requirements defined here. In the following sections, we will describe technical approach we will take to meet our objectives in light of the requirements imposed by these properties.

# 12. OVERVIEW OF THE TECHNICAL APPROACH

Based upon the previous investigations, there are basically two types of environments for parallel processing. As described above, these are:

- Shared memory programming (SMP)
- Networked (Distributed) Computers

Each of these presents different design conditions on the architecture of a parallel processing run-time environment. The conditions for these two environments are presented below.

## SHARED MEMORY PROGRAMMING (SMP)

As described above, the number of processes that can be run concurrently without special validity checks can be no greater than the number of processes scheduled at the same time and priority. However, a majority of simulations will have only one or two processes scheduled at precisely the same time and priority. Yet, as described in the prior sections, they may have a high degree of inherent parallelism. This is due to large numbers of independent models, typically large independent model instances, e.g., of complex radios or switches. Therefore, this category is divided into two general cases as defined below.

### Large Number Of Processes Scheduled At The Same Time

As exemplified by the EPLRS Capacity simulation, SMP is easiest to deal with when large numbers of processes are scheduled at the same time and priority. This leaves only one condition to be resolved - they must share no resources with processes that are currently running. When this check is satisfied, the complete independence of the process removes questions of validity.

### Small Number Of Processes Scheduled At The Same Time

If processes in independent model instances represent independent entities in a real system that are not tied by a master clock, then these processes may be scheduled at different times, even though they are candidates for running concurrently.

To take advantage of the inherent parallelism, we must relax the constraint of being scheduled at the same time with the same priority, and look to other measures to insure validity. As described in Sections 9 and 10, we will impose a limit on the time differential between the current time T, and the scheduled time of the next process in the schedule queue. We refer to this as $\Delta T$. In addition, we must ensure validity. To do this, we must find the maximum value of $\Delta T$ ($\Delta T_{max}$) that can be used while ensuring that validity criteria are still met. Then, processes scheduled within $\Delta T_{max}$ of each other are candidates for running concurrently. However, they must also satisfy the data independence check, i.e., they must share no resources.

## NETWORKED (DISTRIBUTED) COMPUTERS

Networked (distributed) computer simulations are currently being used in Distributed Interactive Simulation (DIS) environments. These simulations generally fall into two categories. In one, the design is tightly coupled to the network and corresponding network timing - a very complicated software synchronization problem that affects model designs. In the other, they are loosely coupled time-wise, and network transit timing is of little concern. We know of no environments that permit one to design separate simulations that have a high degree of interaction time-wise, yet eliminate a designer's concerns for the machine environment while making effective use of networked simulations.

The approach proposed by PSI to implement distributed computer simulations is to synchronize the clocks associated with each simulation to within some $\Delta T_{max}$ to ensure validity of results. The distributed simulations with which we have worked in the past permit a relatively wide range of $\Delta T_{max}$ with a corresponding high degree of concurrent processing. We have designed the protocols required to synchronize the independent simulation clocks. These are described in Section 14.

Another problem in distributed computer simulations is how to share data. This has generally been done by message passing. Message passing is just another way to ensure updating a data structure before it is used. One updates the data structure on the sending side and sends it to the receiving side. When the receiving side gets it, it can be used. The designer ensures data coherency by timing the passing of messages, i.e., the updates must always be there when needed.

In GSS, these protocols can be automated so that the designer need not be concerned with the design of lock-outs, communication between processors, nor insuring that the latest copy of a resource is available when needed. However, it is the system designers job to see that functional specifications are met. Likewise, it is the simulation designers job to see that validity requirements are met.

At run-time, these interface resources must be flagged as interprocessor resources. If a process that shares such a resource is running in one processor, then a process that shares a copy of that resource in another processor must wait until the first process is done before it can proceed. This requires a handshake protocol between processors, and corresponding semaphores and communication facilities, to insure that the latest copy is shipped across the boundary before another process proceeds to use it.

To perform these functions, PSI has designed an interprocessor resource option in GSS that can be used in both SMP and distributed computer simulations. This option automates the protocols required to ensure that, when a process is scheduled, copies of any interprocessor resources will be updated if necessary. This is described in Section 13.

## BASIC CRITERIA FOR CONCURRENT PROCESSING

To summarize, we can take advantage of parallel processing as follows. In an SMP environment we can use the *basic* criteria below.

- Check for processes scheduled at the current time with the current priority. If any exist, they are candidates for parallel processing. If they meet the data independence criteria, assign a thread, e.g., a POSIX thread (p-thread), to run on the allocated processor. Allocate a processor if none is currently allocated.
- If the ΔTmax option is used, check for processes scheduled within ΔTmax of the earliest time - Te (oldest time of currently running processes). If any exist, they are candidates for parallel processing. If they meet the data independence criteria, assign a thread to run on the allocated processor. Allocate a processor if none is currently allocated.

In a distributed computer environment, the ΔTmax option is invoked and we will use the following basic criteria.

- Check the next process in the queue to see if it is scheduled within ΔTmax of the last network synchronization time. If so, and it meets the data independence criteria it is a candidate for concurrent processing.
- Proceed to run it unless it shares an interprocessor resource. If it shares an interprocessor resource, and that resource is not in use, check to see if the latest copy resides on this processor. If not residing there, request the latest copy and proceed.
- If a shared interprocessor resource is in use, wait until it is free, then request the latest copy and proceed.

The above criteria forms the basic approach to our proposed automated protocols for synchronizing simulations running concurrently and sharing data in a distributed computer environment. This approach will support very large scale simulations running concurrently and interchanging information. It will also permit tying the simulation clocks to the real-time clock to insure that simulations run in real-time. This implies that the individual simulations would normally run at least as fast as the real-time clock.

In the next section, we will extend this basic approach to include schedulers in each processor in the SMP environment, and cross-processor scheduling in both the SMP and distributed computer environment. We will also consider checking the module instance of processes to provide for different criteria for interior versus interface processes.

# 13. RUN-TIME ARCHITECTURE CONSIDERATIONS

The run-time architecture required to support an SMP is significantly different from that for distributed computers. In the case of SMP, there is a single operating system controlling the use of each processor. This is described in further detail below.

In the case of distributed computers, each computer has its own operating system. With the proposed design, the platforms need not be the same. Simulations run independently on each processor, using their own scheduler, with two exceptions: (1) a process on one machine can schedule a process on another machine; and (2) a process on one machine can share a resource with a process on another machine. This is described in further detail below.

## SHARED MEMORY PROGRAMMING (SMP) ENVIRONMENT

Figure 13-1 below provides a top level view of the proposed design modifications to the GSS run-time environment for an SMP environment. Note: In recent literature, SMP is used to denote Symetrical Multi-Processor, a similar concept. In addition to the GSS Process Scheduler, there is a Processor Allocator to allocate processes scheduled at the current time (or within $\Delta$Tmax) to the available processors. Referring to the POSIX standard, [13], OS level calls can be used to assign POSIX threads (p-threads) to processors. For our design, a *p-thread* corresponds to a GSS or VSE thread as defined in Section 7, and each processor will likely be in a separate *scheduling allocation domain (a subset of processors in a multi-processor environment)*. This will provide the ability to allocate specific processes (threads) to specific processors.

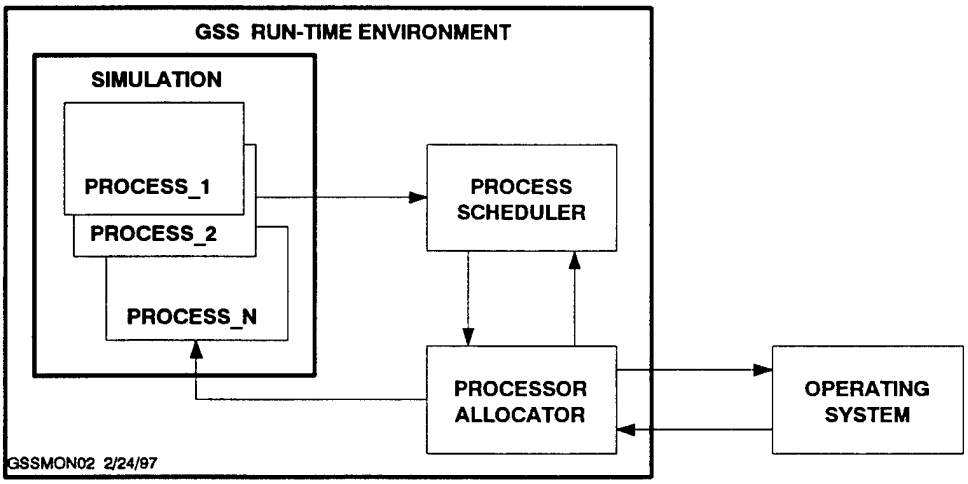


Figure 13-1. GSS Run-Time Environment for an SMP machine.

### Processor Allocation - By Model

The processor allocator must determine if the next process on the schedule queue can be run concurrently with processes already running. The rules for deciding this were outlined in Section 12. It must then select the processor upon which to run that process. This is best decided based upon the model or model instance containing that process. Since models and model instances will be independent via the visual architectural design approach, processes in one model instance typically can run concurrently with processes in a different model instance. Conversely, processes contained within the same model or instance are likely to be scheduled or called sequentially and are best allocated to the same processor. Thus, the processor allocator will initially be allocating the processes within given models or model instances to a given processor. Once they are assigned they will remain there unless there is need to perform load balancing.

### Model Migration To Achieve Balanced Loading

In addition to model grouping, we must consider that, in general, there will be more processes to be run concurrently than there are processors. This implies contention for processor time, and the corresponding desire for load balancing. For example, a processor containing multiple model instances that are very active due to the scenario may be holding up the simulation while other processors sit idle. One would like to migrate one or more of the active model instances to less active processors. This can be done by flagging processes, in those models to be migrated for reallocation when they are next scheduled. Statistics on processor utilization can be kept dynamically to help make migration decisions.

### Master and Slave Schedulers

We have only discussed a single (master) scheduler thus far. In Sections that follow we will show the potential benefits of having separate local schedulers on each processor to support local scheduling without going through the master scheduler. It would then be necessary to interchange schedule information on processes outside the local processor with those of other processors containing the scheduled processes. This would have to be done every time a local scheduler's clock is ready to advance beyond the current time, or beyond the $\Delta$Tmax time if that option is invoked.

As the individual processor clocks advance, they would have to exchange schedules with the processors containing the scheduled processes. This can cause the clocks *not* to advance, since these processes can be scheduled at the current time, or within the $\Delta$Tmax time. Processes scheduled at the same clock time would proceed to run until the local clock time would cause the master clock to advance, possibly beyond $\Delta$Tmax time. Exchange of cross-processor schedules would then occur again, and processes would be run without advancing the clock (at least beyond $\Delta$Tmax time). This cycle will proceed until all cross-processor schedules would cause the master scheduler's clock to advance. A more detailed explanation of cross-processor scheduling and clock synchronization is provided in the next section - Distributed Computers.

## NETWORKED (DISTRIBUTED) COMPUTERS

Distributed computers are used to support concurrent processing of relatively independent subsystems and simulations that have been brought together to examine their dynamic interaction, and the corresponding effects on validity. Good examples include separate simulations of communication systems that are tied together, or separate simulations of communication systems that are tied to other real-time systems. In these cases, the different simulations may have been developed separately and users subsequently recognized the need to bring them together to produce a more accurate representation of the overall system.

The ability to do this has been enhanced in the DoD using Distributed Interactive Simulation (DIS) experiments. PSI has supported clients who have connected four separate simulations to produce a much more realistic picture of what really happens when higher resolution models of the various pieces are brought together. One example was the injection of realistic communication models. In simulation of real-time systems, communication channels are typically represented using stationary statistics (often just perfect channels). Results can be quite optimistic without the additional levels of resolution that are invoked when connecting to a communication network simulation.

When simulations are connected using networks of computers, two significant problems must be resolved in order to maintain validity of results. These are synchronization of otherwise independent simulation clocks, and ensurance of coherent data accesses when data is shared across processor boundaries.

### Synchronization Of Simulation Clocks To The Real-Time Clock

Each computer on a network has its own internal real-time clock. When running a simulation, there is a corresponding simulation clock. If the simulation is to be run in real-time, then the simulation clock will be *paced* by the real-time clock. This implies that the simulation normally runs faster than real-time, but the simulation clock is not allowed to advance ahead of the real-time clock. In this case, the simulation will run close to, but behind, the real-time clock. This can lead to questions of validity.

Another approach is to ensure that the simulation clock does not drift more than some $\Delta T$ from the real-time clock, i.e., either ahead or behind. Then one can experiment with $\Delta T$ to find the $\Delta T_{max}$ that ensures validity of results.

The next question is to synchronize the real-time clocks in each computer. This is typically done by assigning one computer as having the master clock and synchronizing all of the other computer clocks to that one. Since real-time clock drift is normally insignificant compared to the difference in simulation clock times, this is easily accomplished during initialization.

### Synchronization Of Simulation Clocks Independent Of The Real-Time Clock

Finally, we must consider the case where we want the simulations to run as fast as possible, independent of the real-time clock. Note that they may run slower or faster than the real-time clock, possibly by significant multipliers. In this case, we must synchronize the simulation clocks directly across computers.

The general solution to this problem is to insure that none of the individual simulation clocks advances more than a selected $\Delta T$ beyond the others when running concurrently. This ensures that all simulations are within a $\Delta T$ range of each other. To select $\Delta T$, one can experiment to find the $\Delta T_{max}$ that ensures validity of results. Note that $\Delta T_{max}$ can be zero, in which case no simulation clock can advance until all clocks are ready to advance, and the number of concurrent processes that can be run can be no greater than the number of processors with a process scheduled at the same next time.

This approach does not prohibit a simulation from advancing its clock more than the selected $\Delta T_{max}$. It implies that the next clock-time in all simulations must be compared to Te - *the earliest clock-time of all concurrently running simulations.* If the clock-time of the next process in any simulation falls within $Te+\Delta T_{max}$ interval, then that process can run. If it falls outside, then it must wait until it falls within a future $Te+\Delta T_{max}$ interval as it changes, or it becomes the next clock-time across all simulations. In the latter case, the advance could be large compared to $\Delta T_{max}$.

Implementation of this approach requires that each computer be updated with the Te value. This can be done each time Te changes, or each time the next clock-time in all simulations exceeds the current $Te+\Delta T_{max}$ time. The latter would require less frequent updates and therefore less overhead. Selection of the best scheme will require experimentation.

### Cross-processor Scheduling

Depending upon the application, and the design of the simulation, it may be convenient to schedule processes in one processor from another processor. Although this feature may be required, it introduces further questions of validity and corresponding complexity.

As illustrated in Figure 13-2, processor A has scheduled processes residing in itself and also in processor B. Likewise, processor B has scheduled processes residing in itself and also in processor A. We must now consider how the schedules in each processor can be updated with schedule entries of processes that reside in that processor that were scheduled in another processor.

We will start with the case where $\Delta T_{max} = 0$. Then none of the simulation clocks in any processor can be advanced until *all* are ready to advance. When this occurs, one can exchange the cross-processor scheduling information, updating the queues as necessary. This can cause the simulation clock to remain at the same time since processes scheduled from a different processor could have been scheduled NOW (at the current time).

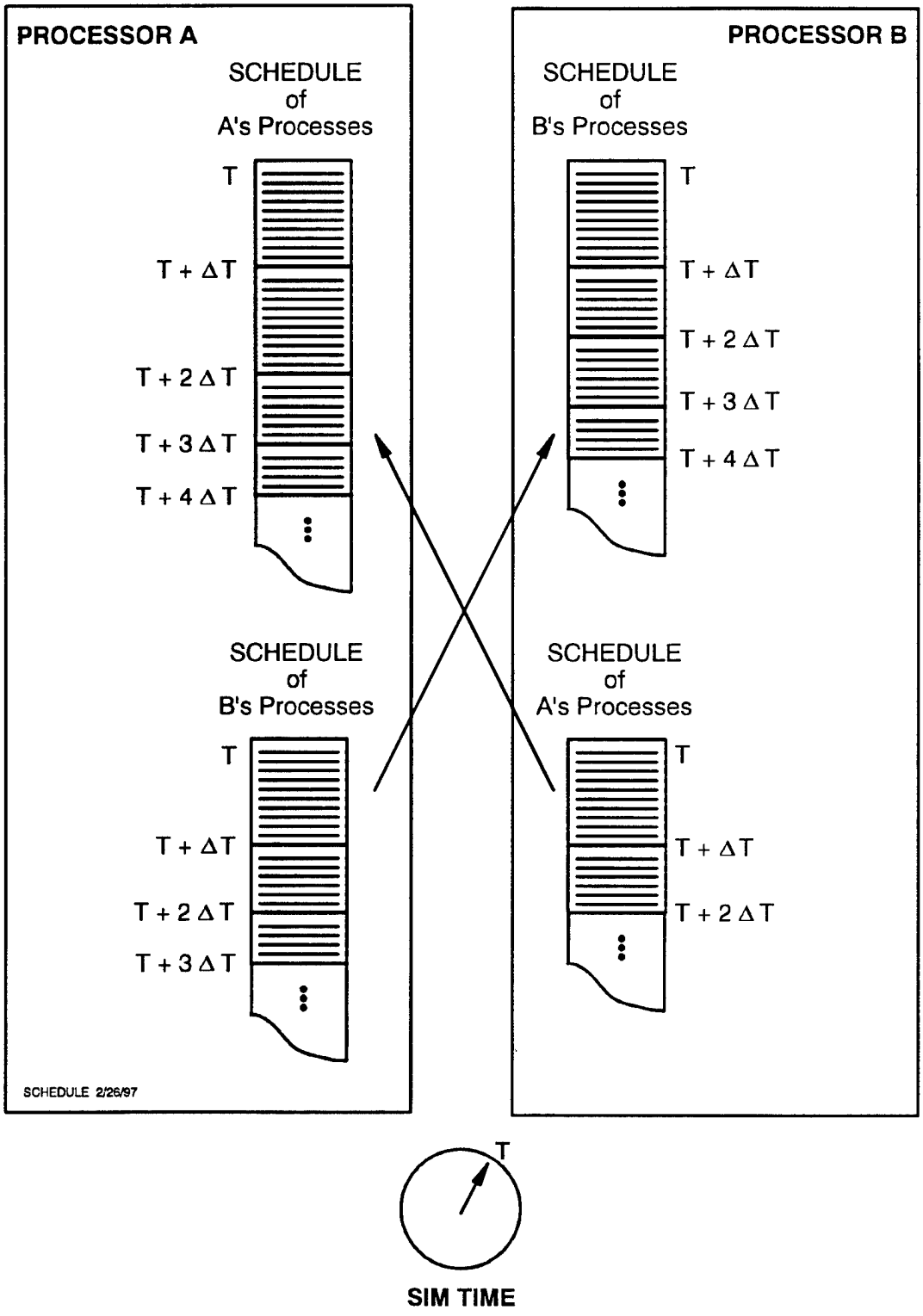


Figure 12. Processes scheduled across processors.

In the case that the simulation clocks are synchronized to within ΔTmax, we must determine when to exchange the cross-processor schedules. By definition, ΔTmax represents an interval during which the activities on one processor are considered independent of the activities on another. At the end of a ΔTmax interval, dependent information is exchanged. This allows the schedulers to advance their clocks independent of each other, as long as the maximum difference between clocks does not exceed ΔTmax from the earliest clock-time, Te, at the start of the interval. Also by definition, the interval ends at Te+ΔTmax.

The outstanding question with this approach is how to deal with the cross-schedules of processes that reside in a processor whose clock has advanced beyond the scheduled time on the process that schedules it. Consider the following cases.

Cross scheduled process schedule times with respect to the processor of interest are:

1. NOW - Current Simulation Time ($T_{SIM}$)
2. IN ΔT TIME FROM NOW ($T_{SIM} + \Delta T$)
3. AT A FUTURE CLOCK-TIME ($T_{AT}$)

In all cases, one must interpret these schedule time as pertaining to the simulation clock in the processor that contains the scheduled process. In case 3, this is an absolute clock-time that can fall behind the simulation clock in the processor containing the scheduled process. In GSS, a simulation clock-time in the past is an invalid event that causes a fatal error.

As indicated at the beginning of this section, cross-processor scheduling introduces further questions of validity and corresponding complexity. If this feature is to be built into GSS, cross-processor schedules should be entered into the queue in terms of an absolute clock-time with respect to the simulation clock on the processor containing the process. When the above three cases are converted to a specified clock time, $T_{AT}$, relative to the clock of the process containing the scheduled process, then for $T_{AT} \geq T_{SIM}$, $T_{AT}$ holds. For $T_{AT} < T_{SIM}$, $T_{AT} \Rightarrow T_{SIM}$.

**Interprocessor Resource Coherence Protocols**

Simulations running concurrently on distributed computers can share data via GSS resources. With the automated facilities described in this section, the modeler is alleviated from having to design message protocols that ensure information is passed in valid time frames.

Figure 13-3 illustrates the situation when resources are shared between two processes that reside in different processors. In this case, resource RES_SA (RES_SB) is shared by PROC_SA in processor A and PROC_SB in processor B. A resource management protocol is required so that when one of these processes runs, it uses the latest version of the resource to ensure coherent use of the data. This implies that if PROC_SA is running, and PROC_SB is scheduled to run, PROC_SB must wait until PROC_SA completes and RES_SA is copied to RES_SB.

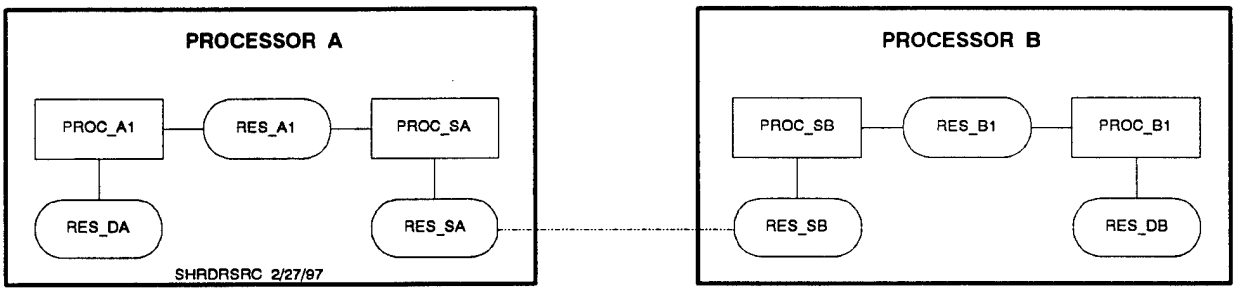


Figure 13-3. Resources shared across processors.

To support this facility, a protocol will be implemented within the GSS Run-Time Monitor that follows the design of one done for prior multi-processing projects by PSI. This protocol tracks resources shared across processors to flag when they are in use, and whether they have the latest copy. When a process that shares one of these resources is scheduled to run, this protocol checks these conditions, determine if the process can be run, and copy the resource if necessary.

## COMMON ARCHITECTURE CONSIDERATIONS

From the above analysis, we see that there are three basic environment cases to be considered when designing a parallel processing run-time architecture. These cases are:

1. SMP with a large number of processes scheduled at the same time
2. SMP with a small number of processes scheduled at the same time
3. Distributed computers

Using GSS, the key protocols and corresponding decision processes required to support these three cases are very similar if not identical. We will examine these below with an eye to developing a common architectural approach.

### Synchronizing Cross-processor Scheduling

In case 2, an SMP environment with a small number of processes scheduled at the same time requires the use of a $\Delta$Tmax window of independence to provide sufficient opportunities for concurrent processing. This is similar to the requirement for distributed computers, case 3. If the protocols for synchronizing cross-processor scheduling for distributed computers were used in the SMP environment, then processes interior to independent models and model instances could be run concurrently using individual schedulers within each processor. This would eliminate the bottleneck of going through a master schedule to schedule every process in the simulation.

As described above, this same protocol could apply to case 1, the SMP environment with a large number of processes scheduled at the same time, with $\Delta Tmax = 0$. Again, this would eliminate the bottleneck of going through a master scheduler to schedule every process in the simulation. It also implies that a general multiple-processor scheduling protocol can apply to all three cases.

### Ensuring Coherent Use Of Interprocessor Resources

The problem of ensuring coherence of accesses to interprocessor resources shared by two or more processes in different processors exists in the SMP environment as well as the networked computer environment. SMP machines have moved toward large local memories that may have built in hardware cache coherency schemes, refer to Frank [1]. We should note that the KSR machine that implemented Frank's design is no longer on the market. Whether or not the coherency protocols are already implemented on a machine is not the issue. What is important is they are needed in all three cases enumerated above. The only difference is that, in a single shared memory environment, or one where the coherency is automated, copying resources may not be required by the protocol. Control over the process scheduling sequence would still be a required part of the GSS Run-Time environment.

### Allocation Of Processes (Threads) To Processors

Given an approach to scheduling processes (threads), e.g., as described above, one must consider the allocation of processes to processors. The allocation process need only occur when:

1. The first process in a model (instance) is selected to run.

2. The first process in a model (instance) is scheduled after the model has been tagged for migration to another processor.

Case 1 above represents the initial allocation of a processor to run the first thread for the selected model or model instance. All subsquent threads of that model (instance) will run on that same processor until it is determined that the model (instance) is to be migrated to another processor.

In case 2, the second bullet, the first thread selected to run after its model (instance) migration decision is made must have a new processor allocated. Normally when a thread is selected to run, it simply runs on the processor that its model currently resides upon.

# 14. TOP-LEVEL RUN-TIME SYSTEM ARCHITECTURE

This section provides a top-level architecture specification for the run-time environment for both an SMP machine and a distributed computer network. The three major run-time architectural elements required for parallel processing are covered; these are the scheduler, the processor allocator, and the interprocessor resource. The protocols required to support the parallel processing environment are described for each of these elements below. Those decision processes required to accommodate differences between an SMP machine and a computer network are described as appropriate.

In addition to modifying the existing GSS and VSE Run-Time environments, the translation systems will also have to be modified to accommodate the new run-time architecture.

## MULTIPLE PROCESSOR SCHEDULER DESIGN

Most simulations built by PSI have a very high degree of inherent parallelism. When an architect follows the visual design rules for model independence, processes that are highly connected are grouped within independent models. Most of the time, there are large numbers of instanced models, e.g., instances of complex node models interconnected in a communication network. Typically, in a large model instance, there are relatively few schedules to processes outside the model instance, and internally scheduled processes can correspond to threads with sequences of many calls. Therefore the processes in an instance are best run together on a separate processor.

Given a relatively high number of schedules within an independent model (instance), and the likelihood that a number of instances will be allocated to a single processor, a local scheduler can be used to schedule these interior processes without going back to the master scheduler. Each local scheduler would run those processes allocated to its processor sequentially. Interface processes in an adjacent model on another processor would have to be scheduled by the master scheduler since they are not independent.

Local schedulers would be similar to the current GSS single processor scheduler, except that they must interface with the master scheduler to get and coordinate updates to the master simulation clock. When pulling the next process from the schedule queue to be run, local schedulers must stay within the $\Delta$Tmax interval to proceed without coordinating with the master scheduler. If the next process to be run is outside the $\Delta$Tmax limit, then the local scheduler must exchange information with the master scheduler before it can proceed. This exchange includes the receipt of cross-schedules from other processors, sending cross-schedules to other processors, and receiving simulation clock and interval control information.

Upon exchanging this information with each processor, the master scheduler can decide whether or not to advance the master clock to the next interval, or to leave the clock as is. If there are any cross-scheduled processes that stay within the interval, then the clock remains as is.

When the next process to be run on *all* schedulers is beyond the current interval, then a new earliest schedule time, Te, is determined and a new interval is initiated.

We will now formulate the rules for a general multi-processor run-time environment, whether SMP or distributed computers, with a scheduler on each processor. We will assume that the following conditions hold:

- A $\Delta$Tmax synchronization interval has been selected to provide a sufficient window of opportunity for concurrent processing while ensuring validity.
- All processors contain a cross-processor schedule queue as well as a local schedule queue.
- All cross-processor schedule queue entries are sent to the proper local schedule queues when the Te + $\Delta$Tmax limit is reached on the local schedule clock.
- When processes are scheduled across processors, those processes scheduled NOW or in $\Delta$T will have their schedule times converted to an absolute time, $T_{AT}$, of the simulation clock in the processor containing the scheduled process. For $T_{AT} \geq T_{SIM}$, $T_{AT}$ holds. For $T_{AT} < T_{SIM}$, $T_{AT} \Rightarrow T_{SIM}$ .

Given the above assumptions, the following rules will be followed by each scheduler.

1. Check the next process in the local schedule queue to see if it is scheduled within $\Delta$Tmax of the start of the current synchronization interval. If so: in an SMP environment, go to 2; in a distributed computer environment, go to 3. Else wait for the next synchronization interval.
2. If the next process thread meets the data independence criteria it is a candidate for concurrent processing; go to 3. Else go back to 1.
3. If the next process thread does not share an interprocessor resource, proceed to run it. If it shares an interprocessor resource, and that resource is in use, go to 4; else check to see if the latest copy resides on this processor. If not residing there, request the latest copy and proceed to run the process.
4. If a shared interprocessor resource is in use, wait until it is free, request the latest copy, and proceed to run the process

# 15. OPTIMAL PROCESSOR ALLOCATION METHODS

As defined above, two processes are independent if they do not share any resources. We also defined *connectivity* as the inverse of independence, such that two scheduled processes have corresponding *threads* that are connected if they, or any processes in their thread *calling trees*, share a resource. Thus processes are independent if and only if their threads are not connected. In VSE or GSS, the connectivity of a module is defined in the architecture environment, not the language environment. Since the VSE and GSS top level monitors maintain this information, a process *connectivity matrix* can be built prior to run time. An illustration is shown in Figure 15-1.

SCHEDULED PROCESSES

| RUNNING PROCS | | P1 | P2 | P3 | P4 | P5 | | Pi | | Pm-1 | Pm |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | P1 | - | 0 | 0 | | | | | | | |
| | P2 | 0 | - | | 0 | | | | | 0 | |
| | P3 | 1 | | - | | 1 | | | | | |
| | P4 | | 0 | | - | 0 | | 0 | | | |
| | P5 | | | 0 | 0 | - | | | | | |
| | | | | | | | | | | | |
| | Pj | | | | | 1 | | 1 | | | 1 |
| | | | | | | | | | - | | |
| | Pn-1 | | | | | | | | | - | 0 |
| | Pn | | | | | | | | | 0 | - |

Figure 15-1. Process connectivity matrix - a snapshot.

This matrix has entries in each row of a column where the processes in those rows share a resource with the process in the column; otherwise they are blank. The more sparse this matrix, the higher the degree of independence of processes in the system.

To illustrate a mechanism for tracking the state of candidacy for running a process concurrently, consider the entries in the matrix in Figure 14. When a process runs, its row is changed from 0's to 1's. When the process terminates, the row is changed from 1's to 0's. Processes P2, P4 and Pm-1 are candidates to be run concurrently with the processes that are currently running (P3 and Pj). Others are blocked because they share resources with the running processes. To determine if a process can be run, one need only check the column corresponding to that process to determine if any entries have the value 1, indicating that there are processes connected to it that are already running, thereby inhibiting it from being run at that time.

From this we can also see that ordering of processes to be run concurrently will affect processor productivity. This problem is similar to the optimal ordering problem associated with optimal sparse matrix reduction described by Berry, [6], and Hachtel [7]. This provides the basis for a potential look-ahead feature. It results from the VSE/GSS shared resource concept that emulates physical systems at the hardware level. Processes can communicate only through shared resources, and the resulting connectivity is defined by the module architecture. This is described further in the Section Optimal Ordering of Scheduled Processes.

To allocate a processor to a process (thread) and run it, the allocator invokes a p-thread call that starts a parallel thread running on a specified processor, refer to [13]. The GSS allocator will track the processors that have been allocated to GSS processes, and can allocate the same processor to all processes within a model (instance).

To perform the determination, the connectivity matrix can be used to show independence, as shown in Figure 14. When a process, RP, is run, its row is scanned changing 0's to 1's. When the process terminates, the row is scanned again changing 1's to 0's.

Processes interior to a module (instance) may be called instead of being scheduled, and are thus invoked directly from the point at which they are called. As part of the scheduled process thread, these calls do not affect the schedule queue, and will serve to substantially increase processor productivity. They will become part of the p-thread assigned to a processor.

In the parallel processor environment, a processor allocator must be used to dispatch multiple processes to run concurrently provided they are independent. It is also possible to have this allocator determine the ordering of processes queued up at time T so as to maximize processor productivity. This is a form of the time-optimal control problem with constrained resources. However, since a process running in time slice T can schedule itself or another process in T, the queue can continue to grow dynamically within the time slice. Therefore, the ordering process must continue in a reduced form within the time slice.

# 16. INTERPROCESSOR RESOURCE PROTOCOL DESIGN

An interprocessor resource protocol must be designed to: (1) track when an interprocessor resource is in use; (2) determine where the latest copy resides; and (3) update a copy when necessary before it is used. More specifically, the following rules apply.

- A copy of an interprocessor resource must reside on each processor that contains a process that shares it.
- When a process that shares an interprocessor resource is running, no other copies may be used.
- Before a process that shares an interprocessor resource can run, an interlocking check must be made that ensures no other process is running that shares it, and a return of *free* implies that it is now locked up for use by only that process.
- Before a process that shares an interprocessor resource can run, the latest copy of the resource must reside in the processor containing that process.

Implementation of this protocol may depend upon the hardware architecture. For example, the Kendall Square Research machine, KSR-1, provided hardware support for local memory access coherency. This was described in the paper by Frank, [1]. When "built-in" facilities are available, they should be used because of the speed advantages they can provide.

## ADDITIONAL SPEED UP FACTORS

There are a number of additional considerations for more effective use of parallel processors to improve speed. We will cover the most attractive of these in this section. All can be implemented downstream, if and when they are deemed desirable.

### Optimal Ordering Of Scheduled Processes

As previously indicated depending upon the order in which processes are run, more or less of them can be run concurrently, affecting the overall running time. Theoretically, processes can be picked in an order that maximizes concurrent processing, and therefore minimizes run time. This will depend, in general, upon the length of time each scheduled process runs, a factor that cannot be predicted accurately.

Ideally, the list of processes scheduled at the current time could be ordered such that the next process that is most independent of the others will be scheduled first. This can be evaluated from a *relative independence measure,* e.g., one that counts the number of resources connected to the process, where each resource can also be weighted in accordance with the number of processes that share it. This measure can be used to determine an effective value for the schedule key, and stored with the model (instance) identifier for every process.

As processes are scheduled at run-time, ordering can take place in a separate processor allocated to the run-time environment shown in Figure 13. In the current version of GSS, the run-time environment determines the ordering of processes based solely upon the times they are scheduled to run and their priority. Ordering is not considered since every process runs sequentially and there is virtually no effect on running times.

As the rows and columns in the connectivity matrix are ordered to show independence, it leads to groupings about the diagonal that show relative independence of the processes. These groupings will normally show up in independent models (instances). Therefore, an optimal ordering can result simply from an architecture that produces independent models (instances). When an architect follows PSI's design rules, this occurs automatically, and processor allocation can be limited to models instead of processes. This is a great simplification.

It is important to note that this simplification is the result of visualization of the architecture. Visualization is the result of separating data from instructions to allow the one-to-one mapping of graphical icons into the actual code. This is what allows an architect to quickly determine the critical independence properties of a design just by visual inspection of the drawings. When the designer follows PSI's design rules, visual inspection of the drawings can also be used to quickly assess the inherent parallelism of a system.

**Load Balancing**

As a task or simulation proceeds, independent modules or model instances will have been assigned to the various processors. As a scenario unfolds, some model instances will be very active and others barely active. When lopsided processor utilization occurs, speed multipliers fall. The solution to this problem is to shift the load so that processor utilization is balanced. This requires: (1) instrumentation for tracking processor utilization; and (2) a protocol for balancing the load.

Most SMP operating environments provide built-in functions for obtaining processor loading information. Given this information, the load can be balanced by migrating model instances from more highly loaded processors to those that are more lightly loaded. This can be accomplished by allocating processes in a particular instance to the new processor as they are scheduled. Similiar steps can be taken in certain distributed computer environments, e.g., with the IBM SP2.

**Multi-Tasking Of The Queue Manager And Processor Allocator**

Scheduling of processes can be done concurrently with continued processing of the simulation. Once a process hands off a schedule command to the process scheduler, it can go ahead and continue to process while the scheduler inserts the process into the schedule. Similarly, the processor allocator can perform its scan of the schedule to allocate processors to GSS processes, including optimal ordering if used.

# 17. USER-INTERFACE DESIGN CONSIDERATIONS

Based upon the prior sections on architecture, we will now consider design to be made to the existing GSS/VSE user-interface environments. The focus is on scheduling and calling model instances.

## INSTANCE POINTER VALUE RULE

To specify an instance from outside an instanced model, the instance values are assigned in an argument list following the process_name in a SCHEDULE or CANCEL statement. The general format for a schedule statement is as follows.

SCHEDULE process_name [INSTANCE instance_pointer_1 [, ..., instance_pointer_n] ]

If a process is scheduled within the same instanced model, then the instance pointers are passed implicitly, and should not appear in the argument list. When a process is executing, the instance pointers defined for the models containing that process hold the current values of the instances that the process represents. These instance pointers can be used to automatically attach the proper resource instance to the process when it runs. These pointers are also available for use by the process in a read-only mode. The values of model instance pointers are blocked from change by processes within that model.

Referring to Figure 15-1, TPS can SCHEDULE LP and the pointers back to the proper instance of TRS are automatically invoked. If LP schedules KP in MODEL_3, then it must explicitly use the form SCHEDULE KP INSTANCE SOURCE, DEST, where SOURCE, DEST can be any properly defined numeric attributes or literals. Note that trying to connect LP directly to KR would not be permitted architecturally, since there is no way for LP to attach to the proper instance of KR when scheduled by TP. KP could schedule LP.

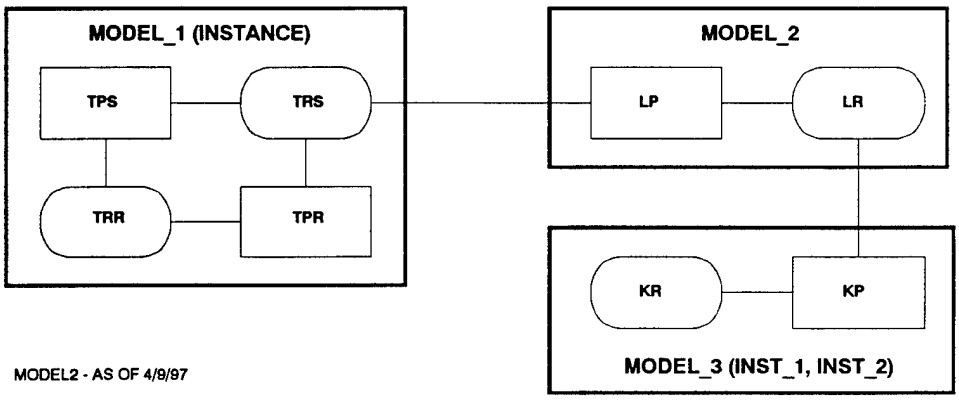


Figure 15-1. Example of interconnections of instanced models.

## CALL STATEMENT RULE

CALL statements are sequential; they cannot be used to increase the number of concurrent processes (parallel paths). They directly control any processes they invoke at the time, rendering them nonindependent from the calling process. Calls from instanced models will automatically carry the current value of the instance pointer to the called process. If independent models are to be run concurrently, they must be scheduled. Calls can serve to invoke a process on another processor, e.g., a utility, but this is not an efficient way to use the processors containing either the process or the call statement. Multiple copies of frequently called utilities would be a more effective solution if they can run in parallel. This represents the typical time memory tradeoff. With memory being relatively inexpensive today, system and simulation designers should look to develop This leads to the desire for utilities that can be copied (instanced) and distinguish them from those that can't.

## GENERAL RULES

Figure 15-2 provides a summary of the cases of interest when using the SCHEDULE and CALL statements to start a thread running in GSS or VSE. A summary of general rules is provided below.

- When a process in an instanced model is scheduled or called, the instance pointers must be specified. The values of the pointers are set as follows:
  - When referenced from a process outside the model, the model instance must be specified as an instance_pointer after the process name.
    Example: SCHEDULE process_name INSTANCE instance_pointer
  - When referenced from a process inside the same model instance, the instance pointer must not appear in the instance_pointer list.
- When a process within an instanced model references another process in that same instance, it automatically invokes the same instance pointers. No arguments are specified relative to the common model instances after the process name.
- References to hierarichal model or other multiple instance pointers must be ordered as specified in the instance pointer list of the process being called. This must be in the order of the hierarchy, from the top down.
- If a process within a hierarchically instanced model is scheduled from outside a subset of the instances, only the new instance pointers must appear in the instance pointer list of the process, in order from the top of the hierarchy down.
- Reuse of the instance-pointer names in resources attached to any process interior to an instanced model must be qualified.

**CASES OF CONCERN**

**Case 1** **SCHEDULEs, CANCELs & CALLs from a noninstanced model to an instanced model.**

Referenced model (process) instances must be identified by specifying a value for the instance pointer, i.e., SCHEDULE process_name INSTANCE instance_pointer.

**Case 2(a)** **SCHEDULEs, CANCELs, & CALLs *within* the same model instance.**

References to the instance pointers of processes within the same instance are implicit, being resolved automatically by the process translator and run-time monitor. Values of the instance pointers of a model are read-only by processes within that model, and cannot be changed by them.

**Case 2(b)** **SCHEDULEs, CANCELs & CALLs *across* instances of the same model.**

References across instances of the same model must be accomplished by using a separate shared interface resource. Direct references are not permitted across different instances of the same model.

**Case 3(a)** **SCHEDULEs, CANCELs, & CALLs from an instanced model to a noninstanced model.**

The instance pointer of the referencing process is passed automatically to the referenced process by the run-time monitor, without any explicit reference, to point back to the resource instances in the referencing model that the referenced process shares with it.

**Case3(b)** **SCHEDULEs CANCELs & CALLs from one instanced model to another instanced model.**

The modeler must identify the referenced process instance by specifying a value for the instance pointer, i.e., SCHEDULE process_name INSTANCE instance_pointer. Pointers to resource instances within the referencing model that are shared with the referenced model are automatically passed to the referenced process.

Figure 15-2. Cases of concern for both the SCHEDULE and CALL statements.

# 18. USING THREADS TO SUPPORT PARALLEL PROCESSING

As described above, the VSE and GSS environments are a complete departure from existing programming languages. They are precisely tailored to visual architectural design, having separated data from instructions as required in a parallel processing environment. This has resulted in three separate languages, the resource language, the process language, and the *simulation* or *task* control specification language, reference the GSS and VSE User Reference Manuals, [11]. The languages contain no abstract software declarations, e.g., what instructions have access to what data, no use of pointers, etc. Users only specify their problem. Architectural specifications are created and maintained graphically, and reside in the database of the architecture environment.

GSS and VSE are thus based on a number of key ideas that lead to large opportunities for parallel computation. By their nature, the design of an apparently sequential program allows blocks of code to be implemented in parallel. There are potentially many independent processes within a single task, as well as many independent tasks. The gains in computational speed can be one to two orders of magnitude.

GSS and VSE, in turn, must be implemented such that the opportunities for parallelism are realized. In particular, we have examined the following question:

> *What language facilities are needed to implement the parallel processing options available through GSS and VSE?*

In the Phase I effort, there were two principle contenders to be analyzed for implementing the parallel architectures resulting from GSS and VSE. The first is a C language version of threads, i.e., parallel POSIX threads (p-threads), refer to [12], and [13]. The second is Ada, refer to [14] and [15]. PSI hired Dr. Henry Ledgard, a prior member of the original Ada language design teams, as a consultant to do an investigation to determine the applicability of Ada.

## TASKS AND THREADS

Multi-tasking is a capability provided by both GSS and VSE to decompose large systems into separately executing programs that can interchange information while they run "concurrently" in a multi-programming environment on a single processor. The main motivation for separate tasks is to break up the software development and support efforts into more manageable pieces. A GSS or VSE task corresponds to a UNIX process which involves a much higher degree of overhead to invoke, compared to a thread. A GSS or VSE process corresponds to a p-thread.

Multi-tasking was also part of the original Ada design and has been improved in the new Ada standard, Ada 95. Ada has a language construct for a task. As being implemented in the new Ada bindings standard, a task in Ada apparently corresponds to a thread, whereas an Ada partition corresponds to a UNIX task.

We must investigate Ada tasks in light of the POSIX standard for POSIX-threads (p-threads) as generally used by parallel processor operating systems. P-threads generally run within a task. VSE and GSS provide a hierarchy of processes within tasks, i.e., we can have thousands of p-threads scheduled within the VSE or GSS run-time environment, all in a single task. We can also have many tasks running concurrently, and communicating easily via intertask resources defined in the architecture environment. This hierarchy is very important from an architectural standpoint, particularly for parallel processing applications.

GSS and VSE tasks (UNIX processes) generally apply to distributed processing whereas p-threads apply to the SMP environment. This is because tasks incur operating system control overhead, and generally take considerable time to invoke relative to the time to run a thread. Figure 18-1 shows the relation between tasks and threads. *Threads* are independent control paths within a task and are generally designed to eliminate this overhead.

POSIX provides the IEEE portable operating system interface for threads, and is an internationally recognized standard. The POSIX standard interface is implemented principally in the C programming language, hence its wide popularity. P-threads, as implemented on parallel processor operating systems, provide the ability to assign a thread to a processor. In a parallel machine environment, many p-threads run under a single task.

P-threads add no direct complexity to the C-programming language; rather, the power of threads lies in the extensive, and somewhat complex, associated library. The price of simplicity of the language model is that the programmer must make all safeguards to guarantee the integrity of a program. However, in the case of GSS or VSE, these assignments can be done automatically. The programmer has neither knowledge nor concern for p-thread assignments.

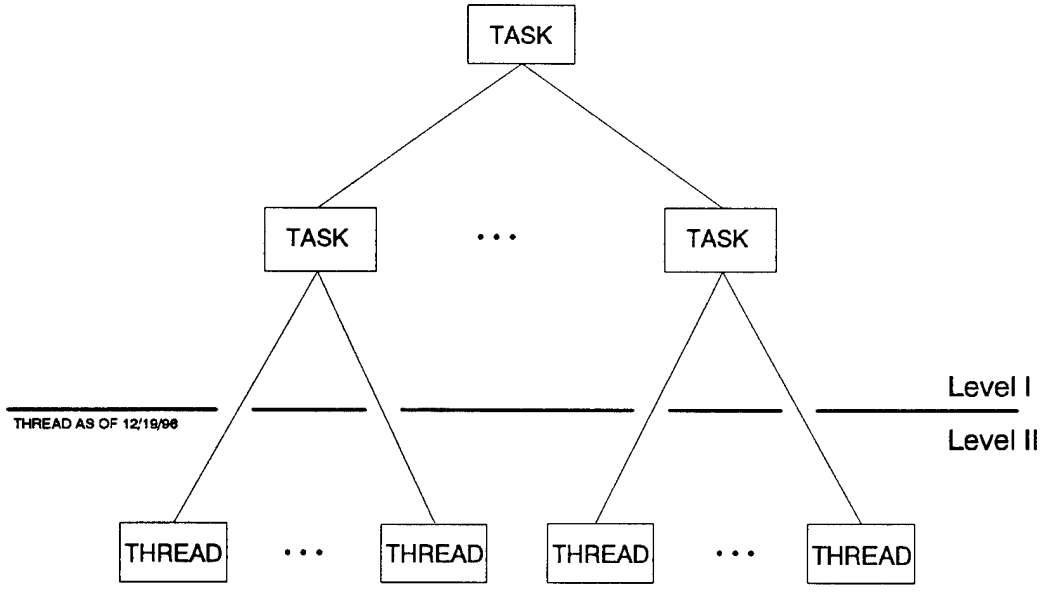


Figure 18-1. Tasks versus threads.

## CODE GENERATION TO SUPPORT CONCURRENT PROCESSING

In this section we will present our findings in comparing Ada to C as candidate languages for code generation from the GSS and VSE language translators. These translators are tasks written in VSE. The code generation portion is a relatively small subsystem compared to the overall GSS (and VSE) systems. Part of this Phase I effort was to analyze and evaluate using Ada in place of C as the generated code for a parallel processing machine environment. The results of this effort are summarized below.

### Assignment Of Threads To Specific Processors

The assignment of specific processors to threads is a critical component in using GSS and VSE for efficient parallel processing. The scheduling algorithms required to achieve the speed multipliers possible with GSS require precise control over processor allocation. In the absence of this control, no guarantees can be made on the outcome. GSS offers potentially huge speed advantages; these advantages can only be realized by an assignment of processors that matches the optimal scheduling algorithms for GSS.

### Using C

The use of threads is common in C implementations on a number of parallel processing platforms. These include Cray, SGI, IBM, SUN, INTEL, and others. This usage is well-established and familiar to many programmers. Threads have well-defined semantics, as defined by use of system parameters and calls to library functions. The facility for threads is a good match for simple concurrent programming. Implementation of threads via C bindings is efficient and commonly understood. The same facility exists in C++, a derivative language that is supplementing the use of C.

The existing POSIX standard defines a specific set of facilities for control of threads. This standard is well-established and is a result of work by the IEEE. The standard is well-documented, well-implemented, and almost universally available.

### Using Ada

The Ada facility for tasks is akin to the C facility for threads. Ada tasking is designed for parallel processing. The Ada language has a good conceptual foundation and strong typing. Thus Ada has a good basis for parallel processing, and a good set of built-in language facilities to support parallel processing.

The Ada language is defined via the "Ada 95" standard (International Standard ISO/IEC 8652:1995 (E)). All newly validated implementations of Ada must conform to this standard. The POSIX standard adds requirements to Ada in order for Ada to conform to the POSIX standard as

well. Ada conforms to the POSIX standard through the use of built-in features, predefined constants and types, and calls to procedures.

However, the bindings for Ada currently make use of the C bindings, because they are universally available. That is, the Ada procedures are defined via calls to the C library. This is a common solution, since the C bindings are almost universally available and well-implemented.

### Universality

C has become both the de facto assembler language and the de facto systems programming language of modern systems. This is because many chip manufacturers hide their proprietary instruction set architectures behind the C compiler. These architectures would be more highly exposed if the machine's assembly language were available directly to programmers.

C, and C++, have become vastly popular languages for code generation. Implementations often use C and C++ to gain efficiency and independence from assembler language. Implementation via C thus ensures portability to most systems.

The C bindings are almost universally implemented. They exist on many computers and many platforms.

C and C++ are ubiquitous. They are highly standardized, most heavily used, and on every machine of interest.

### Summary and Recommendations

The use of Ada for code generation for parallel processing is a conceptually sound solution. However, since the Ada compiler generates C library calls, the direct generation of C is a much more efficient and widely available solution. Given this situation, we believe the best choice for code generation is to stay with C.

# 19. SUMMARY AND CONCLUSIONS

During this Phase I effort, we used the results of our experiments to support the refinement of our approach to parallel processing. Our surprising statistical results caused a reassessment of our originally proposed architectural plan. This required reviewing our visual (graphical) approach to software system architectures, and the manner in which it supports translation into a run-time environment that makes effective use of parallel processors. We then focused upon developing a run-time architecture that offers a unified approach to the Shared Memory Programming (SMP) and distributed computer environments for parallel processing. To that end, we have specified an architecture for multiprocessor scheduling. We also specified the process to processor allocation protocols and the protocols for coherent access to interprocessor resources.

In specifying the process to processor allocation protocols, we looked at measures of merit to compare one approach versus another. To this end, we came up with measures of processor productivity and corresponding measures to predict speed multipliers based upon number of processors used. These measures all stemmed from an effectiveness measure that included the economics of time savings and cost of processor utilization.

As part of the run-time system architecture specification, we considered additional speed-up factors, including optimal ordering of the scheduled processes prior to allocation to processors. However, we have determined that optimal ordering algorithms will not produce any significant results when the architecture of independent models and model instances follows PSI's existing design rules. This is because independent models will be allocated to processors, and therefore run sequentially on that processor. This is further discussed below. Processor-to-processor load balancing, and multi-tasking of the queue manager and processor allocator are additional speed-up facilities that can be added downstream if they are deemed to provide significant improvements to the basic architecture.

Although it was somewhat difficult to sort out where Ada stands with respect to C as a possible contender for code generation, we were able to investigate the state of the POSIX Ada interface/binding standardization effort and the plans for implementation with inputs from Professor Ted Baker of Florida State University. Our position is simply that, since the Ada compiler generates C library calls, the direct generation of C is a much more efficient and widely available solution. Given this situation, we believe the best choice for code generation is to stay with our current C code generation approach, augmented by the POSIX C calls to set scheduling allocation domains (groups of processors) and to make the GSS or VSE thread assignments to our own allocation of processors.

The most surprising result coming out of this Phase I effort is that an optimal ordering of processes can result simply from an architecture that produces independent models (instances). When an architect follows PSI's design rules, this occurs automatically, and processor allocation can be limited to models (instances) instead of processes. This is a great simplification.

It is important to note that this simplification is the result of visual approach to of the architecture. This visualization is the result of separating data from instructions which allows the one-to-one mapping of graphical icons into the actual code. This is what allows an architect to quickly determine the critical independence properties of a design just by visual inspection of the drawings. When the designer follows PSI's design rules, this same visual inspection of the drawings can be used to quickly assess the inherent parallelism of a system.

The approach presented here is revolutionary when compared to existing approaches to building software for parallel processing machines. This is because existing approaches deal at the programming or coding level instead of the architectural level, and most of these approaches are heavily influenced by Object-Oriented Programming (OOP). The OOP approach presents some barriers to improvement in parallel processing. One is the property of encapsulation that keeps data and instructions bound together within an object. The other is the property of inheritance that supports hidden implementation and corresponding interconnections that make it difficult to determine independence automatically. One must rely totally on the programmer to parse his code for separate processors.

By separating data from instructions, we have a one-to-one mapping between the architecture and the code. This provides for separation of architecture from language, allowing the architect to represent the connectivity of data and instructions using a visual graphic approach. It removes the abstract declarations of data sharing from the language, and allows them to be stored in a database for automatic assignment of processes to processors. It also makes everything in GSS or VSE visible to the smallest detail, just by popping the cover off an icon. Pushing down complexity does not depend upon the ability to hide code.

In retrospect, this all seems obvious. But it likely would have never been envisioned within the context of a programming language. PSI's approach was spawned from experience in modeling, simulation, and Computer-Aided Design (CAD) dating back to 1963, and the desire to automate - as much as possible - the engineering design and development process, and now automation of the software development process for parallel machines.

## 20. REFERENCES


[1] Frank, S., "Tightly Coupled Multiprocessor System Speeds Memory Access Times," Electronics, pps 164-169, 1984.

[2] Sietz, C., "The Cosmic Cube," Communications of the ACM, 28-1, pps 22-23, January 1985.

[3] Bell, C. G., "Multis: A New Class of Multiprocessor Computers," Science, Vol. 228, pps 462-467, April 1985.

[4] Bell, C. G., "Ultracomputers A Teraflop Before Its Time," Communications of the ACM, August 1992, Vol.35, No 8.

[5] Daly, W., "The J-Machine: A Fine-Grain Concurrent Computer; MIT VLSI Memo 89-532, May, 1989.

[6] Berry, R., "An Optimal Ordering of Electronic Equations for a Sparse Matrix Solution," IEEE Trans. on Circuit Theory, Vol CT-18, No.1, pps 40-50, January 1971.

[7] Hachtel, G. et al, "The Sparse Tableau Approach to Network Analysis and Design," IEEE Trans. on Circuit Theory, Vol.CT-18, No-1, January 1971.

[8] Rieher, P., "Parallel Simulation Using the Time Warp Operating System," Proceedings of the 1990 Winter Simulation Conference, New Orleans, LS, pp 38-45.

[9] Chandy, K., V. Holmes, and J. Misra, "Distributed Simulation Networks," Computer Networks, Vol. 3, No. 1, pps 105-113, 1979.

[10] Pargus, R. and P. Kambekar, "Guidelines for Dynamic Load Balancing in Conservative Distributed Simulations," Proceedings of the 1990 Winter Simulation Conference, New Orleans, LS, pp 47-52.

[11] GSS & VSE User Reference Manuals, Prediction Systems, Inc., Spring Lake, NJ, 1996.

[12] Steve Kleiman, DeVang Shah, Bart Smaalders. *Programming with Threads,* Sunsoft Press, Sun Micro Systems, Mountain View, CA 1996.

[13] ANSI/IEEE Std 1003.1 (ISO/IEC 9945-1) Second edition 1996-07-12, Information technology - Portable Operating System Interface (POSIX®), Part 1: System Application Program Interface (C Language).

[14] *Ada 95 Reference Manual*, International Standard NCI/ISO/IEC-8652:1995. Published by Intermetrics, Inc. Cambridge, MA, 1995.

[15] *Ada 95 Rationale*, International Standard NCI/ISO/IEC-8652:1995. Published by Intermetrics, Inc. Cambridge, MA, 1995.